UNIVERSIDAD DE SAN CARLOS DE GUATEMALA

FACULTAD DE INGENIERÍA

ESCUELA DE CIENCIAS Y SISTEMAS

LABORATORIO DE BASES DE DATOS 2

SEGUNDO SEMESTRE 2023

PROYECTO 1

GRUPO 18

Nombre	**Carnet**
Javier Alejandro Gutierrez de León	202004765
Enrique Alejandro Pinula Quiñonez	202004707
Gerson Rubén Quiroa del Cid	202000166

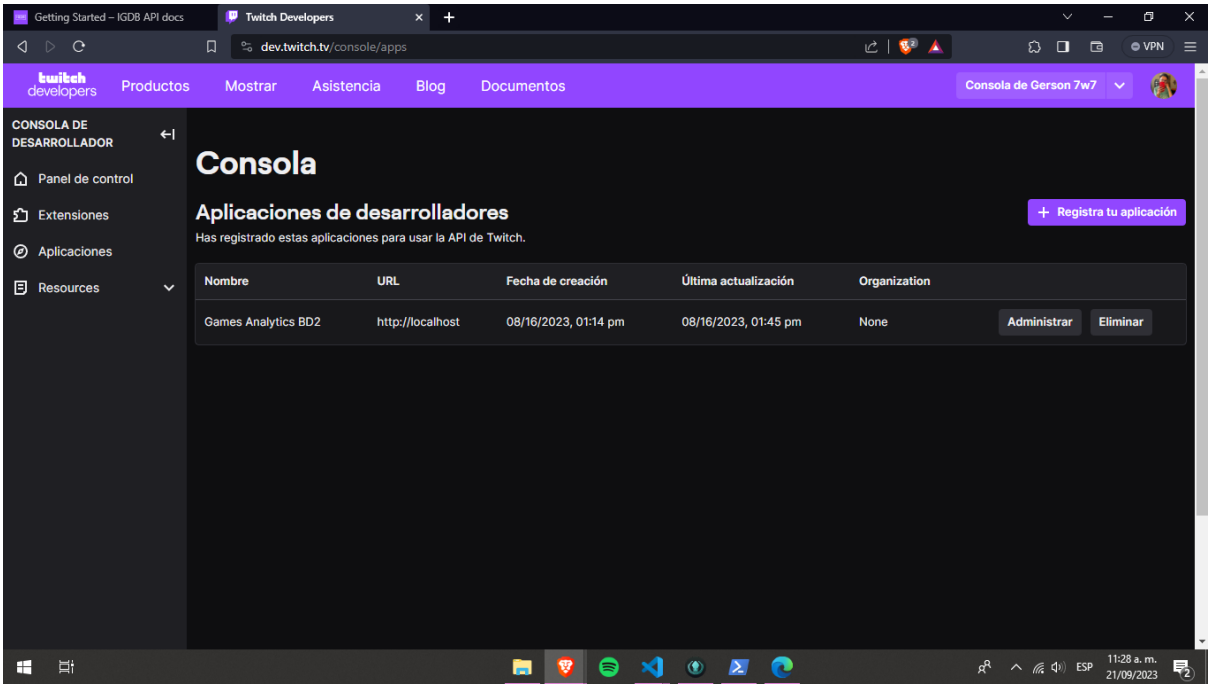
DIAGRAMA ENTIDAD RELACIÓN

El diagrama se modificó al de la entrega del ER ya que nos faltó agregar los Lenguajes Soportados para los juegos. Por lo tanto, se agregaron 3 nuevas tablas siendo estas:

- LanguageSupport: Esta tabla contiene las referencias hacia las otras dos tablas que contienen la información que utilizaremos.
- LanguageSupportType: Esta tabla contiene el tipo de lenguaje que soporta el juego ya sea interfaz, audio o subtítulos.
- Language: Esta tabla contiene el lenguaje soportado con su nombre nativo.

USO DE APIS

Para la extracción de la información de los videojuegos se utilizó una API proporcionada por IGDB.com donde podemos encontrar una infinidad de juegos.
Para utilizar esta API se tuvo que utilizar Twitch, con esto tener autorización para utilizar la API de IGDB. Una vez logueados en nuestra cuenta de Twitch tuvimos que registrar nuestra aplicación en Twitch utilizando nuestra IP local (http://localhost) y dándole un nombre. En este caso le pusimos: Games Analytics BD2.



Luego utilizando Postman, consumimos la ruta de: https://id.twitch.tv/oauth2/token y utilizando nuestro Client ID y Client Secret, estos datos lo podemos encontrar en nuestra aplicación de Twitch.

Esta petición nos regresará el token para que nosotros podamos consumir la API de IGDB y así poder extraer la información sobre los juegos.

games-client.js

Este pequeño programa hecho con node y javascript se utilizó para recuperar los datos de la API de IGDB e insertar dichos datos en nuestra base de datos creada con SQL Server.

Primero configuramos la conexión hacia la base de datos, por seguridad tapamos la clave y la ip de la base de datos:

```
const url = "https://api.igdb.com/v4";
let cont = 198209;
const config = {
  user: "sqlserver",
  password: "[redacted]",
  server: "[redacted]", // Puede ser 'localhost', IP de la BD
  database: "Videojuegos",
  options: {
    encrypt: true,
    trustServerCertificate: true, // Cambiado a false para entorno de producción
  },
};
```

Como podemos observar también se tiene una variable llamada ‘cont’ que nos sirve para llevar la cuenta de los juegos y así no repetirlos y tener la mayor cantidad de videojuegos.

Seguimos con la función ‘obtenerData()’ que esto nos permite hacer fetch hacia la API de IGBD, utilizamos nuestro Client ID y la autorización que nos dió Postman:

```
async function obtenerData() {
  try {
    const response = await fetch(`${url}/games`, {
      method: "POST",
      headers: {
        Accept: "application/json",
        "Client-ID": "[redacted]",
        Authorization: "Bearer [redacted]",
      },
      body: `fields *; limit 500; offset 0; sort id asc; where id >= ${cont};`,
    });

    const data = await response.json();
    return data;
  } catch (error) {
    throw new Error("Error al obtener datos:", error);
  }
}
```

Como se puede observar consumimos la ruta /games para obtener los juegos y con un contador vamos leyendo juegos de 500 cada petición ya que es el máximo permitido. Esta ruta la vamos cambiando dependiendo de lo que se necesite recuperar. La base de la ruta es: https://api.igdb.com/v4 y nosotros agregamos la última parte ya sea /games, /collections, etc para recuperar la información que nos interese.

Tenemos la función 'inserteCollection(data, pool)', esta función recupera las colecciones de los juegos. Para esta función tuvimos que consumir la ruta /collections. Como obtenemos 500 datos, los iteramos e insertamos en la base de datos, primero cerciorándose de que la colección no haya sido insertada anteriormente. La tabla donde se almacenó esta información tiene por nombre Collection.

```
394  async function insertCollection(data, pool) {
395    try {
396      for (const item of data) {
397        const collection = item.id;
398        const name = item.name;
399        const request = new sql.Request(pool);
400        request.input("collection", sql.Int, collection);
401
402        // Verificar si el juego ya existe en la base de datos
403        const checkQuery = `SELECT COUNT(*) AS count FROM Collection WHERE collection = @collection`;
404        const checkResult = await request.query(checkQuery);
405
406        if (checkResult.recordset[0].count === 0) {
407          request.input("name", sql.NVarChar, name);
408
409          const query = `INSERT INTO Collection (collection, name) VALUES (@collection, @name)`;
410          await request.query(query);
411
412          console.log(`Insertado la collecion: ID ${collection}, Nombre ${name}`);
413        } else {
414          console.log(
415            `Colección duplicado encontrado: ID ${collection}, Nombre ${name}`
```

Este mismo proceso se hizo para las demás funciones, cambiando solamente la tabla a insertar datos y la ruta a consumir.

Luego tenemos la función 'insertGame(data, pool)', esta se podría decir que es la función principal ya que es la que se encarga de insertar todos los juegos en la base de datos. Para esta función se consumió la ruta de /games e insertamos en la tabla Game.

```
async function insertGame(data, pool) {
  // 2623
  try {
    for (const item of data) {
      const game = item.id;
      const aggregated_rating = item.aggregated_rating
        ? parseFloat(item.aggregated_rating).toFixed(2)
        : null;
      const aggregated_rating_count = item.aggregated_rating_count
        ? item.aggregated_rating_count
        : null;
      const first_release_date = item.first_release_date
        ? new Date(item.first_release_date * 1000)
        : null;
      const name = item.name ? item.name : null;
      const rating = item.rating ? parseFloat(item.rating).toFixed(2) : null;
      const rating_count = item.rating_count ? item.rating_count : null;
      const storyline = item.storyline ? item.storyline : null;
      const summary = item.summary ? item.summary : null;
      const total_rating = item.total_rating
        ? parseFloat(item.total_rating).toFixed(2)
        : null;
      const total_rating_count = item.total_rating_count
        ? item.total_rating_count
```

Seguimos con la función ‘insertGenre(data, pool)’ para obtener los géneros de cada uno de los videojuegos. En este caso se consumió la ruta de /genres e insertamos en la tabla Genre.

```
async function insertGenre(data, pool) {
  // 0
  try {
    for (const item of data) {
      const genre = item.id;
      const name = item.name ? item.name : null;

      const request = new sql.Request(pool);
      request.input("genre", sql.Int, genre);

      // Verificar si el juego ya existe en la base de datos
      let checkQuery = `SELECT COUNT(*) AS count FROM Genre WHERE genre = @genre`;
      let checkResult = await request.query(checkQuery);

      if (checkResult.recordset[0].count === 0) {
        request.input("name", sql.NVarChar, name);
        const query = `INSERT INTO Genre (genre, name) VALUES (@genre, @name)`;
        await request.query(query);

        console.log(`Insertado el genero: ID ${genre}, Nombre ${name}`);
      } else {
        console.log(`Genero duplicado encontrado: ID ${genre}, Nombre ${name}`);
      }
```

Pasamos a insertar las plataformas de los videojuegos utilizando la ruta de /platforms e insertando los datos en la tabla de Platform.

```
async function insertPlatform(data, pool) {
  // 0
  try {
    for (const item of data) {
      const platform = item.id;
      const name = item.name ? item.name : null;

      const request = new sql.Request(pool);
      request.input("platform", sql.Int, platform);

      // Verificar si el juego ya existe en la base de datos
      let checkQuery = `SELECT COUNT(*) AS count FROM Platform WHERE platform = @platform`;
      let checkResult = await request.query(checkQuery);

      if (checkResult.recordset[0].count === 0) {
        request.input("name", sql.NVarChar, name);
        const query = `INSERT INTO Platform (platform, name) VALUES (@platform, @name)`;
        await request.query(query);

        console.log(`Insertado la plataforma: ID ${genre}, Nombre ${name}`);
      } else {
        console.log(
          `Plataforma duplicado encontrado: ID ${genre}, Nombre ${name}`
        );
      }
```

Por último, tenemos otra función muy importante, ya que son las tablas que unirán la relación entre las tablas que insertamos anteriormente y las que insertamos en los otros archivos js (que se explicarán más adelante) con la tabla principal Game. Esta función lleva por nombre 'insertMuchosAMuchos(data, pool)' ya que como su nombre lo indica, se tuvo que crear varias tablas extras para romper la relación muchos a muchos y así poder relacionar cada tabla con los juegos. Para insertar estos datos tuvimos que consumir de nuevo la ruta /games y leer cada uno de los juegos de nuevo para obtener la información de qué relaciones tienen con las demás tablas. Aquí se insertaron la tablas de:

- InvolvedCompanies: que guardan las relaciones entre juegos y las compañías desarrolladoras.
- Platforms: que guardan las relaciones entre juegos y las plataformas.
- PlayerPerspectives: que guardan las relaciones entre los juegos y las perspectivas de juegos (esta tabla se explicará más adelante).
- GameModes: guardan las relaciones entre los juegos y los modos de juegos (esta tabla se explicará más adelante).

- GameEngines: guardan las relaciones entre los juegos y las tecnologías con que desarrollaron los juegos (esta tabla se explicará más adelante).
- Franchises: aquí se guardan las relaciones entre los juegos y las franquicias de cada juego (esta tabla se explicará más adelante).

```
async function insertMuchosAMuchos(data, pool) {
  // 0
  try {
    for (const item of data) {
      let i = 0;
      const game = item.id;
      const involved_companies = item.involved_companies
        ? item.involved_companies
        : null;
      const platforms = item.platforms ? item.platforms : null;
      const player_perspectives = item.player_perspectives
        ? item.player_perspectives
        : null;
      const game_modes = item.game_modes ? item.game_modes : null;
      const game_engines = item.game_engines ? item.game_engines : null;
      const franchises = item.franchises ? item.franchises : null;

      const request = new sql.Request(pool);
      request.input("game", sql.Int, game);
```

Por último ya, tenemos la función main() y su respectiva llamada que se encarga de darle inicio al programa, realizar la conexión con la base de datos, luego dentro de un ciclo while este irá recorriendo cada juego, colección, plataforma, etc para poder así obtener cada datos y luego insertarlo en su respectivas tablas. Para esto nos ayudará el contador que vimos anteriormente. Por cada consulta que realicemos a la API daremos una espera de 1 segundos para dos cosas: que no se sobrecargue nuestra aplicación y para que podamos realizar otra petición a la API ya que esta nos deja cada 4 segundos. Una vez insertados todos los datos se saldrá del ciclo y terminará la ejecución del programa.

```
async function main() {
  try {
    const pool = await sql.connect(config);

    while (true) {
      const data = await obtenerData();
      if (data.length === 0) break; // Si la lista viene vacía se termina el while
      const exito = await insertMuchosAMuchos(data, pool);
      if (!exito) break;
      cont += 500;

      // Agregar un retraso para no sobrecargar la API
      await new Promise((resolve) => setTimeout(resolve, 1000));
      console.log("cont: ", cont);
    }
    console.log("Se han insertado todos los datos correctamente.");
  } catch (error) {
    console.error("Error principal:", error);
  } finally {
    sql.close();
  }
}

main();
```

funcs_jv.js

Se utilizó la siguiente función para realizar la autenticación con Twitch para obtener el token temporal con el que tenemos permiso de obtener datos de la API

```
async function autenticar() {
    const twitchClientId = '62qm5p8ddoco5cb5jeoz3urq7mhj3r';
    const twitchClientSecret = '8ob453vrwdv9r753ap0vtw210yr0hn';
    const twitchAuthUrl = 'https://id.twitch.tv/oauth2/token';

    const authParams = {
        client_id: twitchClientId,
        client_secret: twitchClientSecret,
        grant_type: 'client_credentials'
    };

    try {
        const response = await axios.post(twitchAuthUrl, authParams);
        twitchAuthData = response.data;
        igdbAccesssToken = twitchAuthData.access_token;
        igdbHeaders = {
            'Client-ID': '62qm5p8ddoco5cb5jeoz3urq7mhj3r',
            'Authorization': `Bearer ${igdbAccesssToken}`
        };
    } catch (error) {
        console.error("Error:", error);
        process.exit(1);
    }
}
```

Para obtener los datos de “game_modes” se utilizó el endpoint de la API : https://api.igdb.com/v4/game_modes. Al ser una tabla con pocos datos solo se realiza la consulta una vez con “fields: *; limit: 500; offset 0; sort id asc; where id >=0;” realizando posteriormente un ciclo con el cual se hacen los inserts de cada registro a la tabla GameMode

```
async function getGameModes() {
    const igdbUrl = 'https://api.igdb.com/v4/game_modes';
    const igdbSearchParams = "fields: *; limit: 500; offset 0; sort id asc; where id >=0;";
    await autenticar();

    try {
        const pool = await sql.connect(config);
        const igdbResponse = await axios.post(igdbUrl, igdbSearchParams, { headers: igdbHeaders });
        if (igdbResponse.status === 200) {
            const games = igdbResponse.data;
            for (const game of games) {
                const request = new sql.Request(pool);
                const query = `INSERT INTO GameMode VALUES (${game.id}, '${game.name}')`;
                await request.query(query);
            }
        } else {
            console.log(`Error al realizar la solicitud a IGDB: ${igdbResponse.status}`);
        }
    } catch (error) {
        console.error("Error:", error);
    } finally {
        sql.close();
    }
    await sleep(4000);
}
```

Para obtener los datos de “game_engines” se utilizó el endpoint de la API : https://api.igdb.com/v4/game_engines. Al ser una tabla con muchos datos se realizó un ciclo el cual utiliza la variable ini con valor inicial 0 utilizando la consulta “fields: *; limit: 500; offset 0; sort id asc; where id >=ini;” realizando posteriormente un ciclo con el cual se hacen los inserts de cada registro a la tabla GameEngines. Al completar el ciclo, obtiene el último id obtenido para empezar por el siguiente en la nueva ejecución del ciclo While, siendo ejecutado de esta forma hasta que el resultado de la consulta sea una lista vacía.

```
async function getGameEngines() {
    const igdbUrl = 'https://api.igdb.com/v4/game_engines';
    let ini = 0;

    while (true) {
        const igdbSearchParams = `fields: *; limit: 500; offset 0; sort id asc; where id >${ini};`;

        try {
            const pool = await sql.connect(config);
            await autenticar();
            const igdbResponse = await axios.post(igdbUrl, igdbSearchParams, { headers: igdbHeaders });

            if (igdbResponse.status === 200) {
                const games = igdbResponse.data;

                if (games.length === 0) {
                    break; // Salir del bucle si no hay más resultados
                }

                for (const game of games) {
                    const request = new sql.Request(pool);
                    request.input('id', sql.Int, game.id);
                    request.input('name', sql.NVarChar, game.name);

                    const query = `INSERT INTO GameEngine VALUES (@id, @name)`;
                    await request.query(query);
                }

                ini = games[games.length - 1].id;
                await sleep(4000);
            } else {
                console.log(`Error al realizar la solicitud a IGDB: ${igdbResponse.status}`);
                break; // Salir del bucle en caso de error
            }
        } catch (error) {
            console.error("Error:", error);
            break; // Salir del bucle en caso de error
        } finally {
            sql.close();
        }
    }
}
```

Para obtener los datos de “franchises” se utilizó el endpoint de la API : https://api.igdb.com/v4/franchises. Al ser una tabla con muchos datos se realizó un ciclo el cual utiliza la variable ini con valor inicial 0 utilizando la consulta “fields: *; limit: 500; offset 0; sort id asc; where id >=ini;” realizando posteriormente un ciclo con el cual se hacen los inserts de cada registro a la tabla Franchises. Al completar el ciclo, obtiene el último id obtenido para empezar por el siguiente en la nueva ejecución del ciclo While, siendo ejecutado de esta forma hasta que el resultado de la consulta sea una lista vacía.

```
async function getFranchises() {
    const igdbUrl = 'https://api.igdb.com/v4/franchises';
    let ini = 0;

    while (true) {
        const igdbSearchParams = `fields: *; limit: 500; offset 0; sort id asc; where id >${ini};`;

        try {
            const pool = await sql.connect(config);
            await autenticar();
            const igdbResponse = await axios.post(igdbUrl, igdbSearchParams, { headers: igdbHeaders });

            if (igdbResponse.status === 200) {
                const games = igdbResponse.data;

                if (games.length === 0) {
                    break; // Salir del bucle si no hay más resultados
                }

                for (const game of games) {
                    const request = new sql.Request(pool);
                    request.input('id', sql.Int, game.id);
                    request.input('name', sql.NVarChar, game.name);

                    const query = `INSERT INTO Franchise VALUES (@id, @name)`;
                    await request.query(query);
                }

                ini = games[games.length - 1].id;
                await sleep(4000);
            } else {
                console.log(`Error al realizar la solicitud a IGDB: ${igdbResponse.status}`);
                break; // Salir del bucle en caso de error
            }
        } catch (error) {
            console.error("Error:", error);
            break; // Salir del bucle en caso de error
        } finally {
            sql.close();
        }
    }
}
```

Para obtener los datos de “companies” se utilizó el endpoint de la API : https://api.igdb.com/v4/companies. Al ser una tabla con muchos datos se realizó un ciclo el cual utiliza la variable ini con valor inicial 0 utilizando la consulta “fields: *; limit: 500; offset 0; sort id asc; where id >=ini;” realizando posteriormente un ciclo con el cual se hacen los inserts de cada registro a la tabla Companies. Al completar el ciclo, obtiene el último id obtenido para empezar por el siguiente en la nueva ejecución del ciclo While, siendo ejecutado de esta forma hasta que el resultado de la consulta sea una lista vacía.

```
async function getCompanies() {
    const igdbUrl = 'https://api.igdb.com/v4/companies';
    let ini = 0;

    while (true) {
        const igdbSearchParams = `fields: *; limit: 500; offset 0; sort id asc; where id >${ini};`;

        try {
            const pool = await sql.connect(config);
            await autenticar();
            const igdbResponse = await axios.post(igdbUrl, igdbSearchParams, { headers: igdbHeaders });

            if (igdbResponse.status === 200) {
                const games = igdbResponse.data;

                if (games.length === 0) {
                    break; // Salir del bucle si no hay más resultados
                }

                for (const game of games) {
                    console.log(game.id + " - " + game.name + " - " + game.description);
                    const request = new sql.Request(pool);

                    request.input('id', sql.Int, game.id);
                    request.input('name', sql.VarChar(255), game.name);
                    request.input('description', sql.Text, game.description || "No hay descripción");

                    const query = `
                        INSERT INTO Company (company, name, description)
                        VALUES (@id, @name, @description)
                    `;
                    await request.query(query);
                }

                ini = games[games.length - 1].id;
                await sleep(4000);
            } else {
                console.log(`Error al realizar la solicitud a IGDB: ${igdbResponse.status}`);
                break; // Salir del bucle en caso de error
            }
        } catch (error) {
            console.error("Error:", error);
            break; // Salir del bucle en caso de error
        } finally {
            sql.close();
        }
    }
}
```

Para obtener los datos de “involved_companies” se utilizó el endpoint de la API : https://api.igdb.com/v4/involved_companies. Al ser una tabla con muchos datos se realizó un ciclo el cual utiliza la variable ini con valor inicial 0 utilizando la consulta “fields: *; limit: 500; offset 0; sort id asc; where id >=ini;” realizando posteriormente un ciclo con el cual se hacen los inserts de cada registro a la tabla InvolvedCompanies. Al completar el ciclo, obtiene el último id obtenido para empezar por el siguiente en la nueva ejecución del ciclo While, siendo ejecutado de esta forma hasta que el resultado de la consulta sea una lista vacía.

```
async function getInvolvedCompanies() {
    const igdbUrl = 'https://api.igdb.com/v4/involved_companies';
    let ini = 230394;

    while (true) {
        const igdbSearchParams = `fields: *; limit: 500; offset 0; sort id asc; where id >${ini};`;

        try {
            const pool = await sql.connect(config);
            await autenticar();
            const igdbResponse = await axios.post(igdbUrl, igdbSearchParams, { headers: igdbHeaders });

            if (igdbResponse.status === 200) {
                const games = igdbResponse.data;

                if (games.length === 0) {
                    break; // Salir del bucle si no hay más resultados
                }

                for (const game of games) {
                    if(game.id == 230398 || game.id == 230400 || game.id == 230401 || game.id == 230403 || game.id == 230427 || game.id == 230428 || game.id == 230434 || game.id == 230435){
                        const request = new sql.Request(pool);
                        console.log(game.id)
                        request.input('id', sql.Int, game.id);
                        request.input('company', sql.Int, game.company);
                        request.input('developer', sql.Char, game.developer ? 'T' : 'F');
                        request.input('porting', sql.Char, game.porting ? 'T' : 'F');
                        request.input('publisher', sql.Char, game.publisher ? 'T' : 'F');
                        request.input('supporting', sql.Char, game.supporting ? 'T' : 'F');

                        const query = `
                            INSERT INTO InvolvedCompany (involved_companie, Company_company, developer, porting, publisher, supporting)
                            VALUES (@id, @company, @developer, @porting, @publisher, @supporting)
                        `;
                        await request.query(query);
                    }
                    /**/

                }

                ini = games[games.length - 1].id;
                await sleep(4000);
            } else {
                console.log(`Error al realizar la solicitud a IGDB: ${igdbResponse.status}`);
                break; // Salir del bucle en caso de error
            }
```

```
        } catch (error) {
            console.error("Error:", error);
            break; // Salir del bucle en caso de error
        } finally {
            sql.close();
        }
    }
}
```

Para obtener los datos de “player_perspectives” se utilizó el endpoint de la API : https://api.igdb.com/v4/player_perspectives Al ser una tabla con muchos datos se realizó un ciclo el cual utiliza la variable ini con valor inicial 0 utilizando la consulta “fields: *; limit: 500; offset 0; sort id asc; where id >=ini;” realizando posteriormente un ciclo con el cual se hacen los inserts de cada registro a la tabla PlayerPerspectives. Al completar el ciclo, obtiene el último id obtenido para empezar por el siguiente en la nueva ejecución del ciclo While, siendo ejecutado de esta forma hasta que el resultado de la consulta sea una lista vacía.

```
async function getPlayerPerspectives() {
    const igdbUrl = 'https://api.igdb.com/v4/player_perspectives';
    let ini = 0;

    while (true) {
        const igdbSearchParams = `fields: *; limit: 500; offset 0; sort id asc; where id >${ini};`;

        try {
            const pool = await sql.connect(config);
            await autenticar();
            const igdbResponse = await axios.post(igdbUrl, igdbSearchParams, { headers: igdbHeaders });

            if (igdbResponse.status === 200) {
                const games = igdbResponse.data;

                if (games.length === 0) {
                    break; // Salir del bucle si no hay más resultados
                }

                for (const game of games) {
                    const request = new sql.Request(pool);
                    request.input('id', sql.Int, game.id);
                    request.input('name', sql.VarChar(255), game.name);

                    const query = `
                        INSERT INTO PlayerPerspective (player_perspective, name)
                        VALUES (@id, @name)
                    `;
                    await request.query(query);
                }

                ini = games[games.length - 1].id;
                await sleep(4000);
            } else {
                console.log(`Error al realizar la solicitud a IGDB: ${igdbResponse.status}`);
                break; // Salir del bucle en caso de error
            }
        } catch (error) {
            console.error("Error:", error);
            break; // Salir del bucle en caso de error
        } finally {
            sql.close();
        }
    }
}
```

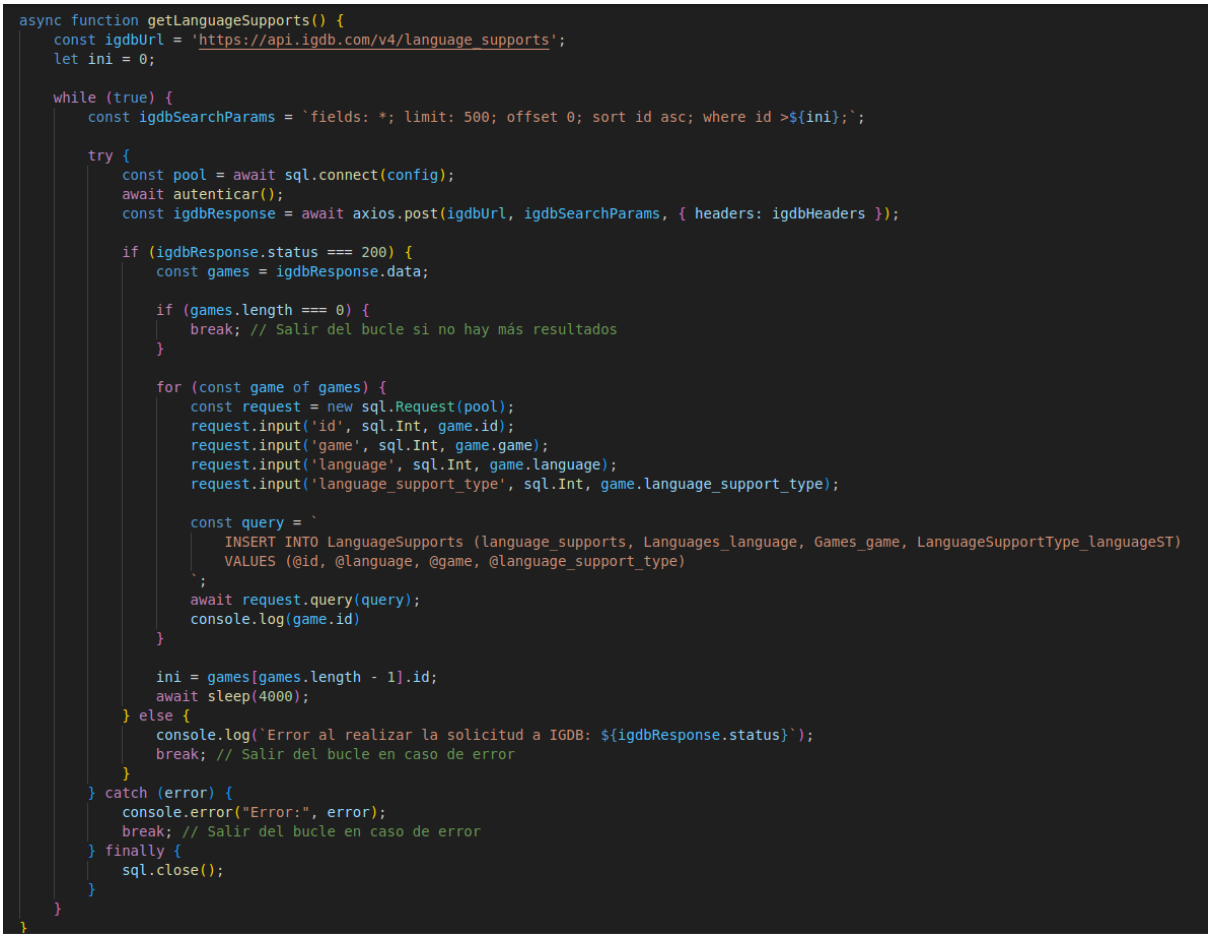
Para obtener los datos de “languages” se utilizó el endpoint de la API : https://api.igdb.com/v4/languages. Al ser una tabla con muchos datos se realizó un ciclo el cual utiliza la variable ini con valor inicial 0 utilizando la consulta “fields: *; limit: 500; offset 0; sort id asc; where id >=ini;” realizando posteriormente un ciclo con el cual se hacen los inserts de cada registro a la tabla Languages. Al completar el ciclo, obtiene el último id obtenido para empezar por el siguiente en la nueva ejecución del ciclo While, siendo ejecutado de esta forma hasta que el resultado de la consulta sea una lista vacía.

```
async function getLanguages() {
    const igdbUrl = 'https://api.igdb.com/v4/languages';
    let ini = 0;

    while (true) {
        const igdbSearchParams = `fields: *; limit: 500; offset 0; sort id asc; where id >${ini};`;

        try {
            const pool = await sql.connect(config);
            await autenticar();
            const igdbResponse = await axios.post(igdbUrl, igdbSearchParams, { headers: igdbHeaders });

            if (igdbResponse.status === 200) {
                const games = igdbResponse.data;

                if (games.length === 0) {
                    break; // Salir del bucle si no hay más resultados
                }

                for (const game of games) {
                    const request = new sql.Request(pool);
                    request.input('id', sql.Int, game.id);
                    request.input('name', sql.VarChar(255), game.name);
                    request.input('native_name', sql.VarChar(255), game.native_name);

                    const query = `
                        INSERT INTO Languages (language, name, native_name)
                        VALUES (@id, @name, @native_name)
                    `;
                    await request.query(query);
                    console.log(game.id)
                }

                ini = games[games.length - 1].id;
                await sleep(4000);
            } else {
                console.log(`Error al realizar la solicitud a IGDB: ${igdbResponse.status}`);
                break; // Salir del bucle en caso de error
            }
        } catch (error) {
            console.error("Error:", error);
            break; // Salir del bucle en caso de error
        } finally {
            sql.close();
        }
    }
}
```

Para obtener los datos de “language_support_types” se utilizó el endpoint de la API : https://api.igdb.com/v4/language_support_types. Al ser una tabla con muchos datos se realizó un ciclo el cual utiliza la variable ini con valor inicial 0 utilizando la consulta “fields: *; limit: 500; offset 0; sort id asc; where id >=ini;” realizando posteriormente un ciclo con el cual se hacen los inserts de cada registro a la tabla LanguageSupportTypes. Al completar el ciclo, obtiene el último id obtenido para empezar por el siguiente en la nueva ejecución del ciclo While, siendo ejecutado de esta forma hasta que el resultado de la consulta sea una lista vacía.

```
async function getLanguageSupportType() {
    const igdbUrl = 'https://api.igdb.com/v4/language_support_types';
    let ini = 0;

    while (true) {
        const igdbSearchParams = `fields: *; limit: 500; offset 0; sort id asc; where id >${ini};`;

        try {
            const pool = await sql.connect(config);
            await autenticar();
            const igdbResponse = await axios.post(igdbUrl, igdbSearchParams, { headers: igdbHeaders });

            if (igdbResponse.status === 200) {
                const games = igdbResponse.data;

                if (games.length === 0) {
                    break; // Salir del bucle si no hay más resultados
                }

                for (const game of games) {
                    const request = new sql.Request(pool);
                    request.input('id', sql.Int, game.id);
                    request.input('name', sql.VarChar(100), game.name);

                    const query = `
                        INSERT INTO LanguageSupportType (language_support_type, name)
                        VALUES (@id, @name)
                    `;
                    await request.query(query);
                    console.log(game.id)
                }

                ini = games[games.length - 1].id;
                await sleep(4000);
            } else {
                console.log(`Error al realizar la solicitud a IGDB: ${igdbResponse.status}`);
                break; // Salir del bucle en caso de error
            }
        } catch (error) {
            console.error("Error:", error);
            break; // Salir del bucle en caso de error
        } finally {
            sql.close();
        }
    }
}
```

Para obtener los datos de “language_supports” se utilizó el endpoint de la API : https://api.igdb.com/v4/language_supports. Al ser una tabla con muchos datos se realizó un ciclo el cual utiliza la variable ini con valor inicial 0 utilizando la consulta “fields: *; limit: 500; offset 0; sort id asc; where id >=ini;” realizando posteriormente un ciclo con el cual se hacen los inserts de cada registro a la tabla LanguageSupports. Al completar el ciclo, obtiene el último id obtenido para empezar por el siguiente en la nueva ejecución del ciclo While, siendo ejecutado de esta forma hasta que el resultado de la consulta sea una lista vacía.



```
async function getLanguageSupports() {
    const igdbUrl = 'https://api.igdb.com/v4/language_supports';
    let ini = 0;

    while (true) {
        const igdbSearchParams = `fields: *; limit: 500; offset 0; sort id asc; where id >${ini};`;

        try {
            const pool = await sql.connect(config);
            await autenticar();
            const igdbResponse = await axios.post(igdbUrl, igdbSearchParams, { headers: igdbHeaders });

            if (igdbResponse.status === 200) {
                const games = igdbResponse.data;

                if (games.length === 0) {
                    break; // Salir del bucle si no hay más resultados
                }

                for (const game of games) {
                    const request = new sql.Request(pool);
                    request.input('id', sql.Int, game.id);
                    request.input('game', sql.Int, game.game);
                    request.input('language', sql.Int, game.language);
                    request.input('language_support_type', sql.Int, game.language_support_type);

                    const query = `
                        INSERT INTO LanguageSupports (language_supports, Languages_language, Games_game, LanguageSupportType_languageST)
                        VALUES (@id, @language, @game, @language_support_type)
                    `;
                    await request.query(query);
                    console.log(game.id)
                }

                ini = games[games.length - 1].id;
                await sleep(4000);
            } else {
                console.log(`Error al realizar la solicitud a IGDB: ${igdbResponse.status}`);
                break; // Salir del bucle en caso de error
            }
        } catch (error) {
            console.error("Error:", error);
            break; // Salir del bucle en caso de error
        } finally {
            sql.close();
        }
    }
}
```

games-features.js

Esta aplicación funciona de manera similar a games-client.js, se cuenta con la misma conexión a la base de datos y la función de obtenerData para recolectar los datos de la API de juegos.

Un ejemplo de las funciones que procesan la información obtenida por la función obtenerData es la siguiente:

```
async function insertTheme(data, pool) {
    try {
        for (const item of data) {
            const theme = item.id;
            const name = item.name
            const request = new sql.Request(pool);
            request.input('theme', sql.Int, theme);
            request.input('name', sql.VarChar, name);

            const query = `INSERT INTO Theme (theme, name) VALUES (@theme, @name)`;
            await request.query(query);

            cont = theme;
        }
    } catch (error) {
        console.error('Error al insertar datos:', error);
    }
}
```

En esta función se recorre el arreglo de datos y se obtienen los valores necesarios, para luego insertarse en la base de datos. Luego se actualizaba el contador con el ID del elemento que se estaba trabajando, en este caso los temas de juegos.

De forma similar se implementaron las funciones para obtener fechas de lanzamiento, géneros de juego, regiones, localizaciones y nombres alternativos.

En la parte de la función main se tiene lo siguiente:

```
async function main() {
    try {
        const pool = await sql.connect(config);
        while (true) {

            //-------------- Theme ---------------
            const data = await obtenerData('themes', 'name');
```

Aquí se crea la conexión a la base de datos y luego, dentro de un ciclo while, se llama a la función correspondiente para obtener los datos, a esta se le pasa como

primer parámetro la ruta que se le quiere agregar a la dirección base para acceder a la API de juegos, luego los parámetros que se quieren obtener de dicha dirección.

Luego de obtener la información se tiene lo siguiente:

Aquí se llama a la función respectiva para insertar los datos y al terminar de ejecutarse dicha función se incrementa el contador en 1, para continuar avanzando según el ID de los elementos.

Por último se agrega un retardo de 300 milisegundos para no saturar las peticiones a la API.