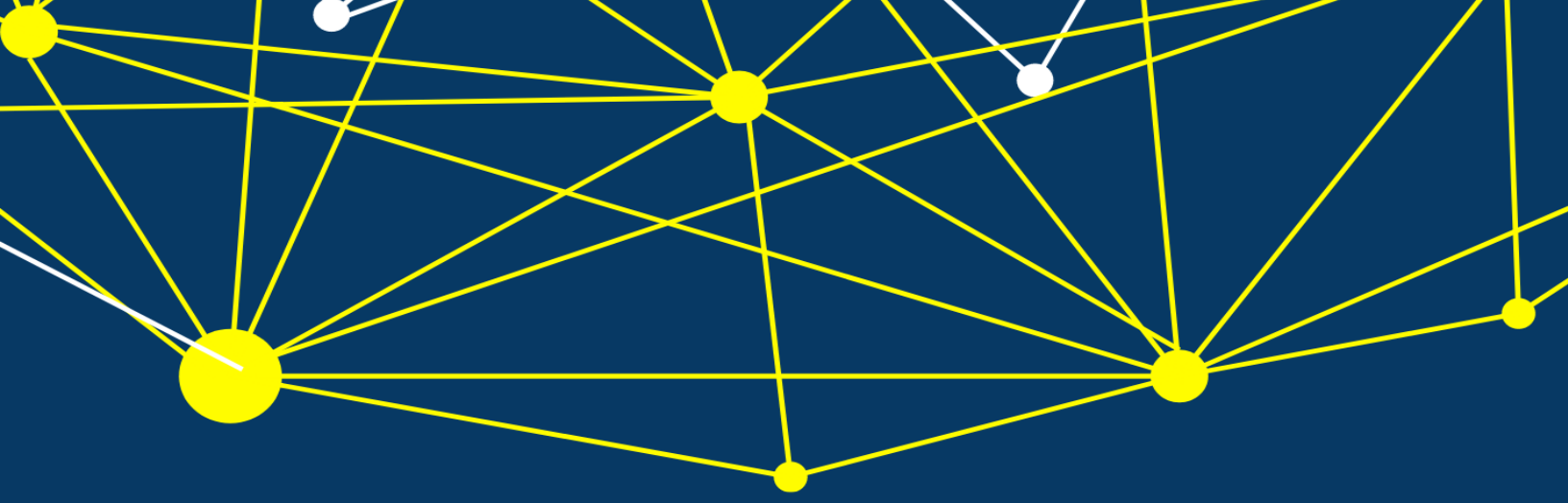

Paradigmas Avanzados de Programación, 3ºGII

Daniel de Heras Zorita
Adrián Borges Cano

PECL3: Cloud

ÍNDICE

Tabla de contenido

Introducción

Este documento ha sido creado para facilitar la explicación del programa desarrollado en Scala, un lenguaje de programación funcional (basado principalmente en la recursión) y orientado a objetos (permite dividir los elementos participes de la lógica del programa en distintas clases). Dicho programa ha sido posteriormente integrado con Azure Cloud, un SaaS (Software As A Service) que nos permite usar su base datos (entre muchas otras funcionalidades) para guardar distintos aspectos del programa.

La práctica trata sobre la realización de un juego llamado 'Cundy Crosh Soga', una adaptación al famoso juego de Smartphone 'Candy Crush Saga' integrado con Cloud. Este trabajo llevado a cabo por Daniel de Heras Zorita y Adrián Borges Cano, corresponde a la PECL3 de Paradigmas Avanzados de Programación, de 3ºGII de la UAH, cuya entrega se ha realizado en mayo de 2023. En este documento explicaremos las distintas versiones que hemos debido generar para cumplir con lo requerido, al igual que una descripción clara sobre el funcionamiento de cada una de las funciones realizadas, en caso de que el lector no sepa cuál es su propósito y quiera conocerlo. Finalmente, haremos una breve pasada por la ejecución del programa en sus distintas versiones.

Antes de comenzar, debemos aclarar que en este documento solo se encontrarán las implementaciones realizadas con respecto a la práctica anterior, ya que todos los detalles a tratar sobre la misma se encuentran perfectamente explicados en la memoria de la PECL2. Sin más dilación, comencemos a adentrarnos en el proyecto.

Versiones del Proyecto

A diferencia de las versiones anteriores, en esta ocasión solo encontramos una versión del proyecto. Esto se debe a que la integración Cloud en nuestra aplicación ha sido directamente realizada en la interfaz gráfica que desarrollamos como parte extra en la práctica anterior. Hemos tomado esta decisión porque deducimos que el lector comprenderá que, si hemos sido capaces de integrarlo en un entorno gráfico, también lo seremos de realizarlo por consola, ya que esta última manera es mucho más sencilla.

Implementación básica

Modificaciones de puntuación

Debemos mencionar que, comparado con la parte anterior, en esta ocasión ampliar el funcionamiento del programa para que devuelva la puntuación obtenida cada vez que realizamos un movimiento ha sido bastante sencillo y ágil de implementar. Lo que hemos tenido que hacer ha sido crear una variable en la interfaz llamada *numPuntos*. Dicha variable suma y almacena progresivamente los puntos obtenidos en cada jugada (obtenidos gracias a *contarPuntuacion*), imprimiéndolos por pantalla. Finalmente, cuando el usuario termina la partida, conoce los puntos finales que ha conseguido, al igual que el tiempo que ha durado su partida.
De cara a la posterior implementación Cloud, se ha incluido un *pop-up* en el que se pide al usuario su nombre de usuario.

Integración Cloud

Para conectar nuestro programa con una base de datos remota, hemos hecho uso del servicio de Azure, donde nos proporcionaba no solo los recursos necesarios para implementar lo que queríamos, sino también material de apoyo para aprender cómo hacerlo, aunque este último en ocasiones ha ocasionado más complicaciones que ayuda.

Para poder enviar la información desde el programa hasta la base de datos, hemos tenido que crear un objeto del tipo JSON que contiene los siguientes datos: id, nombre del usuario, la puntuación obtenida y la duración de la partida. Consideramos que no es necesario enviar la fecha de dicha partida, ya que esta puede ser obtenida directamente desde el servidor cuando recibe la información. Además, de esta forma conseguimos que el paquete enviado sea más ligero puesto que tiene menor número de datos que mandar.

Como bien sabemos de asignaturas previas, en toda base de datos debe haber una columna de la tabla que permita identificar unívocamente a una tupla. Por tanto, para nuestra aplicación, teniendo en cuenta que dicho identificador será asignado en el lado del cliente, hemos decidido que este número sea la suma del día y el mes del año en el que se está jugando, junto con los segundos y minutos que marquen en la hora de dicho momento. De esta forma, podemos asegurar que ningún *id* va a estar repetido, y por tanto cada registro en la base de datos es completamente distinguible del resto.

Para realizar la conexión entre el programa de Scala y la web desplegada desde Azure, utilizamos librerías que contiene la interfaz de Java que permiten establecer conexiones *HTTP*, y una vez esta ha sido establecida, indicamos que queremos hacer un *POST*, para después enviar el paquete.
Adicionalmente, hemos hecho uso de la clase *Desktop* para hacer que la URL donde se muestra la tabla de la base de datos se abra automáticamente una vez hayamos enviado los datos.

Arquitectura Cloud

A pesar de que la arquitectura que tenemos montada en Azure sea muy simple, puesto que el único uso que hacemos de esta es de una base de datos y un par de páginas web, lo hemos enfocado desde un enfoque de arquitectura **microservicios**, de forma que tenemos la plataforma dividida en distintos módulos. En caso de que alguno de ellos falle, esto no supondrá un desastre total en el servicio.

Como se puede observar en la figura de debajo, la aplicación accede a las 2 webs que tenemos en uso. Primeramente, accede a la web de la base de datos para enviar la información que se quiere escribir en la misma, para después poder mostrar de una manera más intuitiva los resultados de las puntuaciones en la web con la interfaz. No debemos olvidar que ambas dependen del funcionamiento de la base de datos, de forma que, si esta última se cae, entonces no se podrá acceder al servicio Cloud. Sin embargo, si por ejemplo se encuentra un fallo en la web de la base de datos, el usuario no podrá guardar la información de su partida, pero igualmente podrá acceder a la tabla clasificatoria sin ningún problema.

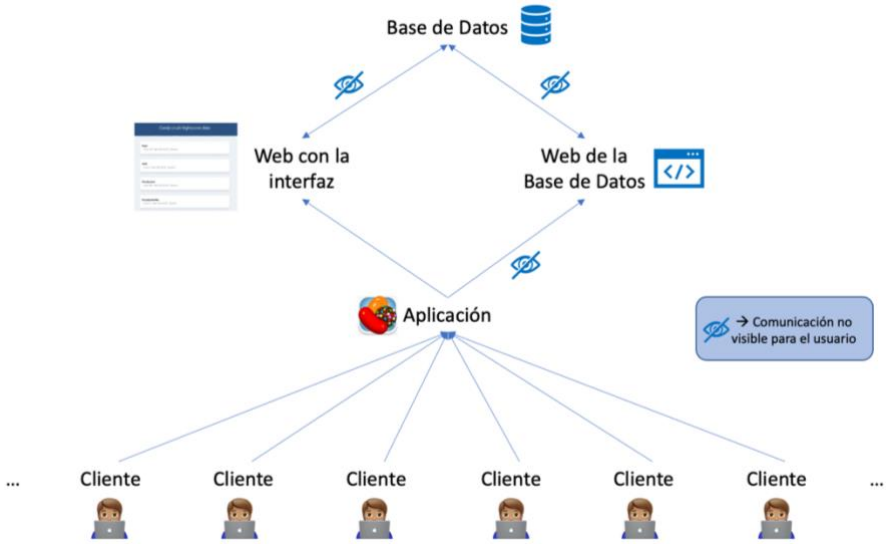

Figura 1: Arquitectura Cloud

Configuración del entorno Cloud

Aplicación de Express.js

La aplicación de Express.js utilizará la biblioteca de *Sequelize* en *JavaScript* para establecer una conexión con nuestra base de datos. La aplicación web se encargará de gestionar peticiones *HTTP POST* y *GET* para insertar datos en la base de datos y para obtener entradas de la tabla respectivamente.

Tendremos un objeto como molde para las entradas de la tabla:

```
const Scores = sequelize.define('scores', {
    id: { type: Sequelize.INTEGER, allowNull: false, primaryKey: true},
    name: { type: Sequelize.STRING, allowNull: false },
    score: { type: Sequelize.INTEGER },
    segundos: { type: Sequelize.INTEGER },
    date: { type: Sequelize.DATEONLY, defaultValue: Sequelize.NOW }
}, {
    freezeTableName: true,
    timestamps: false
});
```

Figura 2: Formato de la tabla

Y tendremos la configuración para lidiar con los mensajes *POST*:

```
app.post('/scores', async (req, res) => {
try {
    const newItem = new Scores(req.body)
    await newItem.save()
    res.json({ scores: newItem })
} catch(error) {
    console.error(error)
}})
```

Figura 3: Formato de los mensajes POST

Donde crearemos un objeto a partir del modelo anterior con los datos del JSON que se mandará en el *POST*, y lo guardaremos en la base de datos. Finalmente, tenemos la configuración para los mensajes *GET*:

```
app.get('/scores/:id', async (req, res) => {
   const id = req.params.id
   try {
  const scores = await Scores.findAll({
      attributes: ['id', 'name', 'score', 'segundos','date'],
      where: {
        id: id
      }})
      res.json({ scores })
   } catch(error) {
       console.error(error)
}})
```

Figura 4: Formato de los mensajes GET

Esto nos permitirá acceder mediante el navegador a la consulta de entradas individuales de la tabla de *scores* en forma de objetos JSON, lo que resulta útil para realizar *debugging*.

Aplicación de Next.js

En esta aplicación web estática, usaremos el cliente de Prisma para conectarnos a la base de datos y mostrar las entradas de la tabla scores en una página web usando un archivo *.tsx*, donde mezclaremos sintaxis de *TypeScript* junto a *HTML* para la presentación al usuario.

Tendremos el modelo de las entradas de la base de datos en el archivo *schema.prisma*:

```
// This is your Prisma schema file,
// learn more about it in the docs: https://pris.ly/d/prisma-schema

datasource db {
  provider = "postgresql"
  url      = env("DATABASE_URL")
}

generator client {
  provider = "prisma-client-js"
}

model scores {
  id        Int
  name      String
  score     Int
  segundos  Int
  date      DateTime
}
```

Figura 5: Archivo *schema.prisma*

En el código de la página web *index.tsx* accederemos a la base de datos obteniendo todas las entradas de *scores*:

```
export const getServerSideProps = async ({ req }) => {
  const scores = await prisma.scores.findMany({
    orderBy: {
      date: 'desc'
    }
  })
  return { props: {scores} }
}
```

Figura 6: Acceso a los datos

Para después usar *HTML* para sacarlos por pantalla:

```
<div
  className="flex bg-white shadow-lg rounded-lg mx-2 md:mx-auto mb-10 max-w-2xl"
  key={name}
>
<div className="flex items-start px-4 py-6">
  <div className="">
    <div className="inline items-center justify-between">
      <h2 className="text-lg font-bold text-gray-900 -mt-1">
        {name}
      </h2>
      <small className="ml-3 text-gray-700 text-sm">
        Score: {score}
      </small>
      <small className="ml-3 text-gray-700 text-sm">
        Date: {date.toString().substring(0,10)}
      </small>
```

Figura 7: Código de los datos con *HTML*

Problemas con la creación y conexión de la base de datos

Aunque en esta memoria se plasme el resultado final de la integración Cloud, dando a entender que ha sido un proceso rápido y sencillo, esto no tiene nada que ver con la realidad. Por ello, a continuación, comentaremos por encima los distintos inconvenientes con los que nos hemos encontrado durante el proceso de desarrollo.

Aplicación de Express.js

Hemos corregido los errores en la dirección de *Sequelize* de nuestra base de datos.

```
const sequelize = new Sequelize('postgresql://Estudiantepap%40pl3-pap:Azure_pap@pl3-pap.postgres.database.azure.com:5432/highscores')
```

Figura 8: comando *sequelize*

De forma que no diese problemas al realizar la conexión y pueda acceder a ella. Podemos realizar un *request POST* con un JSON incluido.

```
de [ ~/pl3 ]$ curl --header "Content-Type: application/json" \
--request POST \
--data '{"id":2,"name":"Fran","score":500}' \
http://scoresWebAppPl3.azurewebsites.net/scores
de [ ~/pl3 ]$
```

Figura 9: POST desde la terminal

Y a continuación, realizar el *GET* en una nueva pestaña de navegador para obtener los valores que acabamos de introducir con la ID:

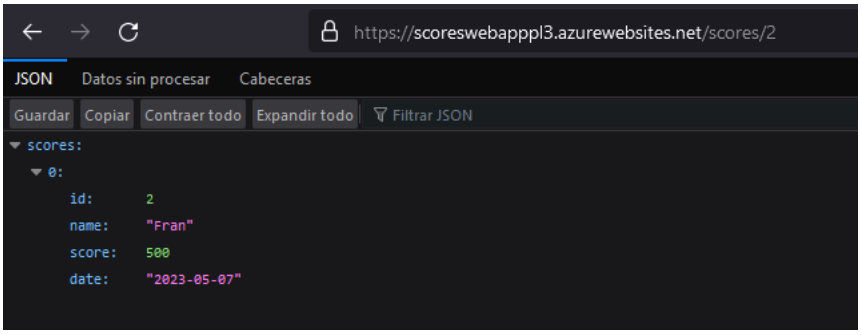

Figura 10: Resultado del GET

Aplicación de next.js

Hemos encontrado múltiples errores para desplegar la aplicación. Al desplegar mediante *git push*, obtenemos un error que informa que no encuentra la versión de *npm* 8.11 para gestionar el despliegue.

```
remote: Unable to locate npm version 8.11.0
remote: The node.js application will run with the default node.js version 16.16.0.
remote: An error has occurred during web site deployment.
remote:
remote: Deployment Failed.
remote: Error - Changes committed to remote repository but deployment to website failed.
```

Figura 11: Error mostrado

Para subsanar esto, forzamos en *package.json* a que se use la versión 8.19 de *npm*, añadiendo un apartado de *engines* en el archivo con las versiones que se van a usar.

```
{
  "name": "next.js-postgresql",
  "version": "0.1.0",
  "engines": {
    "node" : "16.16.0",
    "npm": "8.19.2"
    },
  "private": true,
```

Figura 12: Modificación del JSON

Con estos cambios, ahora se realiza correctamente el despliegue de la aplicación web.

```
remote: Selected node.js version 16.16.0. Use package.json file to choose a different version.
remote: Selected npm version 8.19.2
remote: Updating iisnode.yml at C:\home\site\wwwroot\iisnode.yml
remote: npm WARN config production Use `--omit=dev` instead.
remote: ..........
remote:
remote: up to date, audited 789 packages in 18s
remote:
remote: 41 packages are looking for funding
remote:   run `npm fund` for details
remote:
remote: 23 vulnerabilities (6 moderate, 10 high, 7 critical)
remote:
remote: To address all issues (including breaking changes), run:
remote:   npm audit fix --force
remote:
remote: Run `npm audit` for details.
remote: Finished successfully.
remote: Running post deployment command(s)...
remote: Triggering recycle (preview mode disabled).
remote: Deployment successful.
```

Figura 13: Prueba de despliegue

Sin embargo, cuando intentamos ejecutar 'npm run-script build' en el Azure App Service Editor, nos encontramos con este error:

```
Error: Cannot find module 'C:\Program Files (x86)\npm\8.11.0\node_modules\npm\bin\npm-cli.js'

at Function.Module._resolveFilename (node:internal/modules/cjs/loader:933:15)
at Function.Module._load (node:internal/modules/cjs/loader:778:27)
at Function.executeUserEntryPoint [as runMain] (node:internal/modules/run_main:77:12)
at node:internal/main/run_main_module:17:47
{ code: 'MODULE_NOT_FOUND',
requireStack: []
```

Figura 14: Traza de error en 'build'

Podemos ver que la estancia de la aplicación web de Windows Server también usa el *npm* versión 8.11, y al parecer no lo encuentra en su sistema de archivos. Vamos al apartado de consola en la aplicación web para poder acceder a todos los archivos del sistema.

Figura 15: Consola de Azure

Y comprobamos las instalaciones de *npm*.

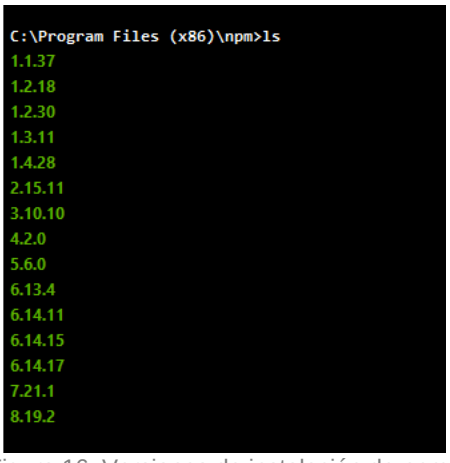

Figura 16: Versiones de instalación de npm

Podemos comprobar que la versión 8.11 de *npm* no está instalada en nuestra instancia de Windows donde se ejecuta la aplicación web. Tras buscar por internet, hemos encontrado una solución sencilla: podemos forzar a la aplicación web a usar la versión 8.19.2 que está instalada añadiendo una regla de configuración.

Nombre	Valor	Origen
WEBSITE_NODE_DEFAULT_VERSION	~16	App Service
WEBSITE_NPM_DEFAULT_VERSION	8.19.2	App Service

Figura 17: Configuración para el uso de npm 8.11

De esta forma, ya no recibimos el mensaje de error sobre la versión de *npm* en el App Service Editor, y podemos ejecutar *npm* y compilar.

```
Page                                                    Size     First Load JS
┌ λ /                                                   3.35 kB        64.4 kB
├   /_app                                               0 B              61 kB
└ o /404                                                3.44 kB        64.5 kB
+ First Load JS shared by all                           61 kB
  ├ chunks/e851e5d3.8f6f5b.js                           68 B
  ├ chunks/f6078781a05fe1bcb0902d23dbbb2662c8d200b3.9ddc08.js  10.4 kB
  ├ chunks/framework.cb05d5.js                          39.9 kB
  ├ chunks/main.f117dc.js                               9.44 kB
  ├ chunks/pages/_app.96d910.js                         484 B
  ├ chunks/webpack.e06743.js                            751 B
  └ css/de79ef46fc63f32047e8.css                        175 kB

λ  (Server)  server-side renders at runtime (uses getInitialProps or getServerSideProps)
o  (Static)  automatically rendered as static HTML (uses no initial props)
●  (SSG)     automatically generated as static HTML + JSON (uses getStaticProps)
   (ISR)     incremental static regeneration (uses revalidate in getStaticProps)

[37;40mnpm [36;40mnotice
[37;40mnpm [36;40mnotice New major version of npm available! 8.19.2 -> 9.6.6
[37;40mnpm [36;40mnotice Changelog: https://github.com/npm/cli/releases/tag/v9.6.6
[37;40mnpm [36;40mnotice Run npm install -g npm@9.6.6 to update!
[37;40mnpm [36;40mnotice
```

Figura 18: Compilado de la web

Sin embargo, al intentar acceder a la web recibimos:

Figura 19: Error de compilado

Si accedemos a los logs de la aplicación web podemos ver lo siguiente.

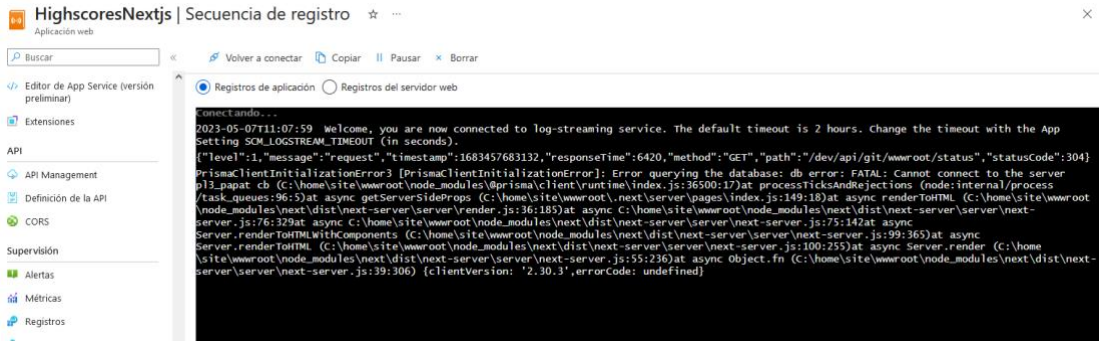

Figura 20: Traza de error desde la consola

Se recibe un error *undefined* que no permite conectarse a la base de datos. Para solucionar esto, hemos reescrito la dirección de la base de datos usando el comando *sed* de la terminal de Azure, aunque la dirección queda igual. Tras esto, la página se conecta correctamente.

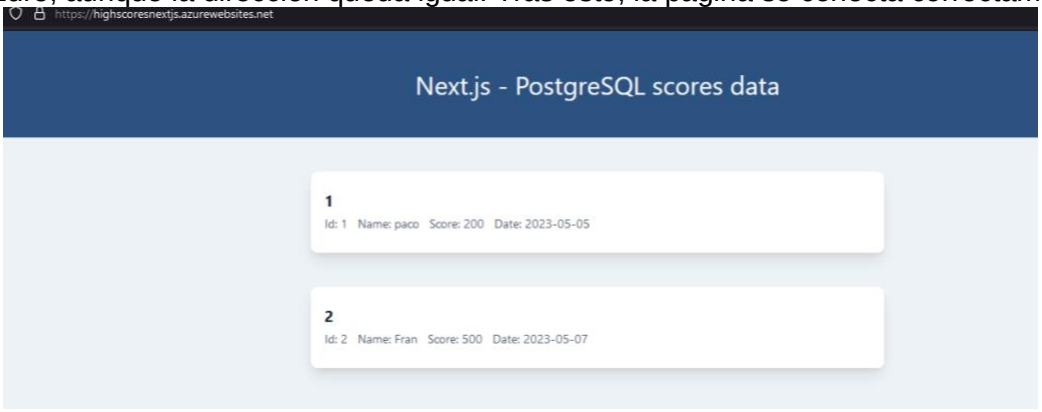

Figura 21: Impresión final de la web

Con esto, finalmente conseguimos arreglar los problemas existentes.

Métodos Implementados

Como hemos comentado en previas ocasiones, únicamente explicaremos la ampliación de lo realizado en la práctica anterior. Por lo tanto, este apartado será mucho más breve en comparación al contenido que pudo llegar a tener en proyectos pasados.

- contarPuntuacion: devolver el número de puntos que obtendrá el usuario dependiendo del bloque que haya seleccionado. Para ello, se le pasa como parámetro de entrada la matriz que resulte después de haber eliminado los bloques

y el bloque que dicho usuario seleccionó antes de eliminarlo en el tablero. Por lo tanto, se devolverá la suma total de los bloques eliminados (ya que se suma 1 punto por cada bloque borrado), más la división entera de dichos bloques entre 10 (porque cada 10 bloques se consigue un punto más). Finalmente, en caso de que sea un caramelo especial, se les añadirá a los puntos obtenidos un valor extra (5 si es una bomba, 10 si es TNT, y 15 si es un rompecabezas). Una vez se han sumado estas 3 partes, se devuelve el número de puntos obtenido.
Cabe mencionar que esta función NO es un contador global del juego, sino que solo cuenta cuantos puntos ha conseguido el jugador en un movimiento.

Ejecución del programa

Explicaremos la ejecución del programa a partir de la selección del modo de juego, que es la parte que más nos incumbe. Como hemos comentado en la ampliación del proyecto, hemos incluido una nueva característica del juego que es la puntuación obtenida. Dicha puntuación se mostrará en ambos modos de juego.

En caso de ejecutarlo en modo automático, la ventana se mostrará de esta forma.

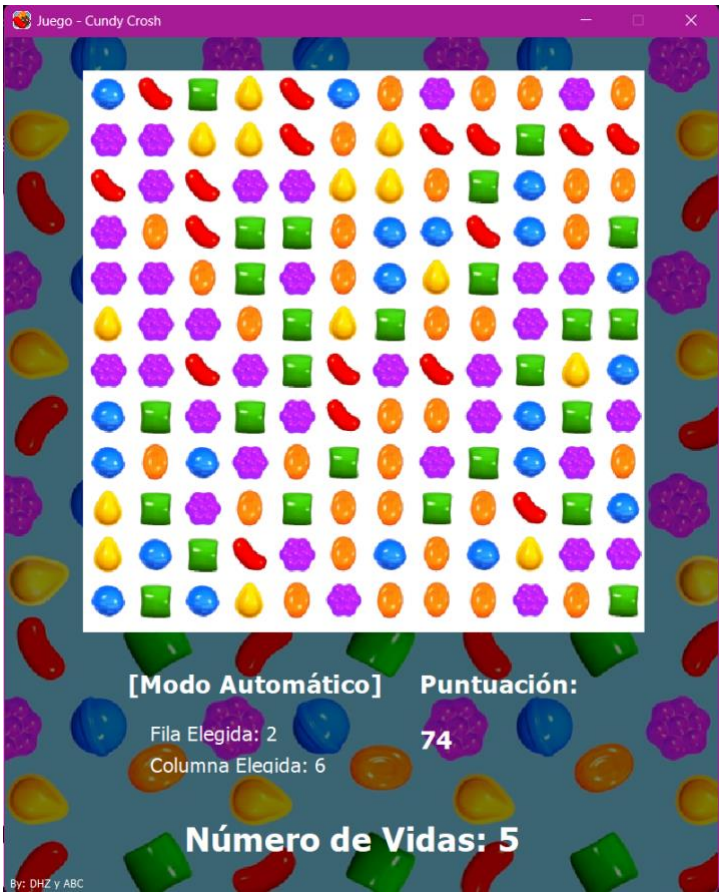

Figura 22: Ventana en modo automático con la puntuación

En caso de ejecutarlo en modo manual, se mostrará esta otra.

Figura 23: Ventana en modo manual con la puntuación

Además, cuando terminemos la partida, en el mensaje de *Game Over* se mostrará la duración de la partida junto con los puntos obtenidos.

Figura 24: Ventana de Game Over

Al pulsar en *OK*, llegaremos a otra ventana emergente, donde en esta ocasión se pedirá nuestro nombre de jugador.

Figura 25: Ventana en la que el usuario introduce su nombre

Una vez el usuario ha introducido su nombre, el programa terminará y dejará de ejecutarse, pero antes de eso abrirá el navegador por omisión del usuario, y abrirá la web que contiene la tabla con los registros clasificatorios de la base de datos.

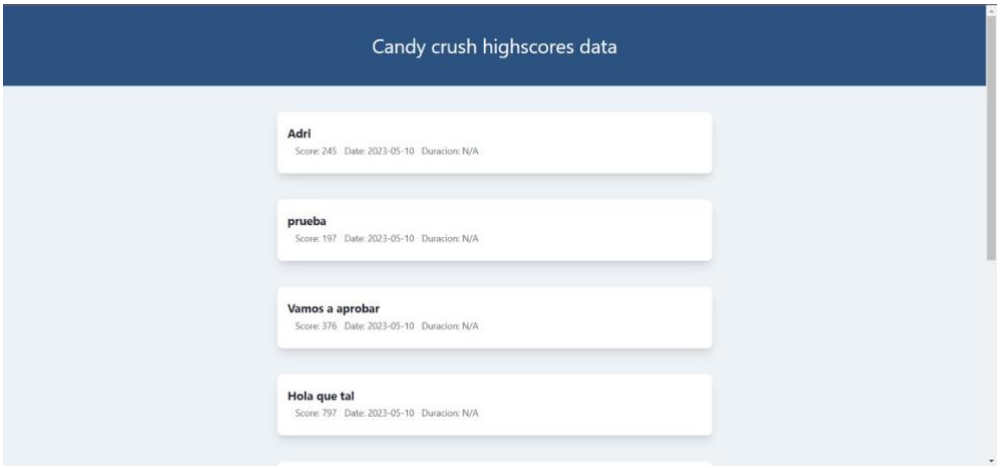

Figura 26: Visor web de la clasificación ordenado de más reciente a más antiguo

Al igual que se hizo con la práctica anterior en Scala, en esta entrega querríamos haber diseñado la interfaz web de una manera personalizada, de forma que continuase con los colores y el estilo implementados en la interfaz del juego.
Desgraciadamente, debido a la falta de tiempo y el estrés con el que hemos tenido que lidiar durante esta práctica, al haber tenido que compaginarlo con el examen de teoría de esta misma asignatura, junto con el resto de asignaturas de la carrera; no nos ha sido posible poder realizar dicha interfaz gráfica como hubiésemos querido, ya que el resultado

final habría sido *chapucero*, y lejos de ensuciar la entrega del trabajo, hemos preferido presentarlo simple pero sólido.

Conclusiones

Finalmente, este transcurso de proyectos en los que se desarrolla un Candy Crush ha terminado. Gracias a este trabajo, hemos sido capaces de ampliar la funcionalidad propia del juego de caramelos, además de poner en marcha una base de datos web en Azure, conociendo el funcionamiento interno de este, y configurando sus distintos parámetros. Esperamos que el lector haya disfrutado y comprendido todo lo desarrollado durante el proyecto, y que todo haya quedado lo más claro posible en esta práctica en la que hemos hecho una integración Cloud.

Anexo: Código de Java para la conexión Cloud

Para poder configurar correctamente la conexión con el servidor y enviar la información que requiere para que esta sea guardada en la base de datos, y mostrada al usuario; hemos tenido que desarrollar el siguiente código en la interfaz del programa, que conforma el lado del cliente.

Se debe tener en cuenta que *idUsuario* es el identificador unívoco creado previamente; *nombreJugador* es el nombre que ha escrito el usuario en el pop-up que se le ha mostrado; *numPuntos* es la puntuación obtenida por el usuario durante la partida; y *segTranscurridos* es la duración de la partida en segundos. El resto de variables son creadas localmente para realizar la conexión.

```
//IMPLEMENTACIÓN DE LA INTEGRACIÓN CLOUD:
// Creamos el Payload del JSON que vamos a mandar
System.out.println("USERID: "+idUsuario);
String jsonPayload = "{\"id\":" + idUsuario + ",\"name\":\"" +
nombreJugador + "\",\"score\":" + numPuntos + ",\"segundos\":" + (int)
segTranscurridos + "}";
System.out.println(jsonPayload);
// Cogemos la URL y realizamos la conexion
URL url = null;
try {
    url = new URL("https://webapppl3.azurewebsites.net/scores/");
    HttpURLConnection conn = (HttpURLConnection) url.openConnection();
    // Indicamos que debe ser un POST
    conn.setRequestMethod("POST");
    // Indicamos que el tad es JSON
    conn.setRequestProperty("Content-Type", "application/json");
    // Enviamos el JSON
    conn.setDoOutput(true);
    try (OutputStream os = conn.getOutputStream()) {
        byte[] input = jsonPayload.getBytes(StandardCharsets.UTF_8);
        os.write(input, 0, input.length);
    }
```

```
    // Imprimimos la respuesta
    int responseCode = conn.getResponseCode();
    System.out.println("Response Code: " + responseCode);

} catch (MalformedURLException e) {
    throw new RuntimeException(e);
} catch (ProtocolException e) {
    throw new RuntimeException(e);
} catch (IOException e) {
    throw new RuntimeException(e);
}

//Hacemos que se abra la página web automáticamente
try {
    java.net.URI uri = new
URI("https://highscoresnextjs.azurewebsites.net/scores/"+idUsuario);
    Desktop.getDesktop().browse(uri);
} catch (URISyntaxException e) {
    throw new RuntimeException(e);
} catch (IOException e) {
    throw new RuntimeException(e);
}
```