

# IBEX35

**Nombre:** Marcos Prego Salvador

**Enlace GitHub:** [Repositorio](#)

<b>Introducción</b>	<b>1</b>
<b>Frontend</b>	<b>2</b>
• DRAG & DROP	2
• LocalStorage	4
• Llamadas Fetchs	4
<b>API</b>	<b>7</b>
• Login y Registro. Autenticación JWT	7
• Controladores	9
• Rutas api	11
• Modelos	11
<b>Docker</b>	<b>12</b>
• Docker Compose	15
<b>Generador de Datos</b>	<b>17</b>
<b>Interfaz Web</b>	<b>18</b>
• Login Modal	19
• Página principal	19
• Página consultas	20

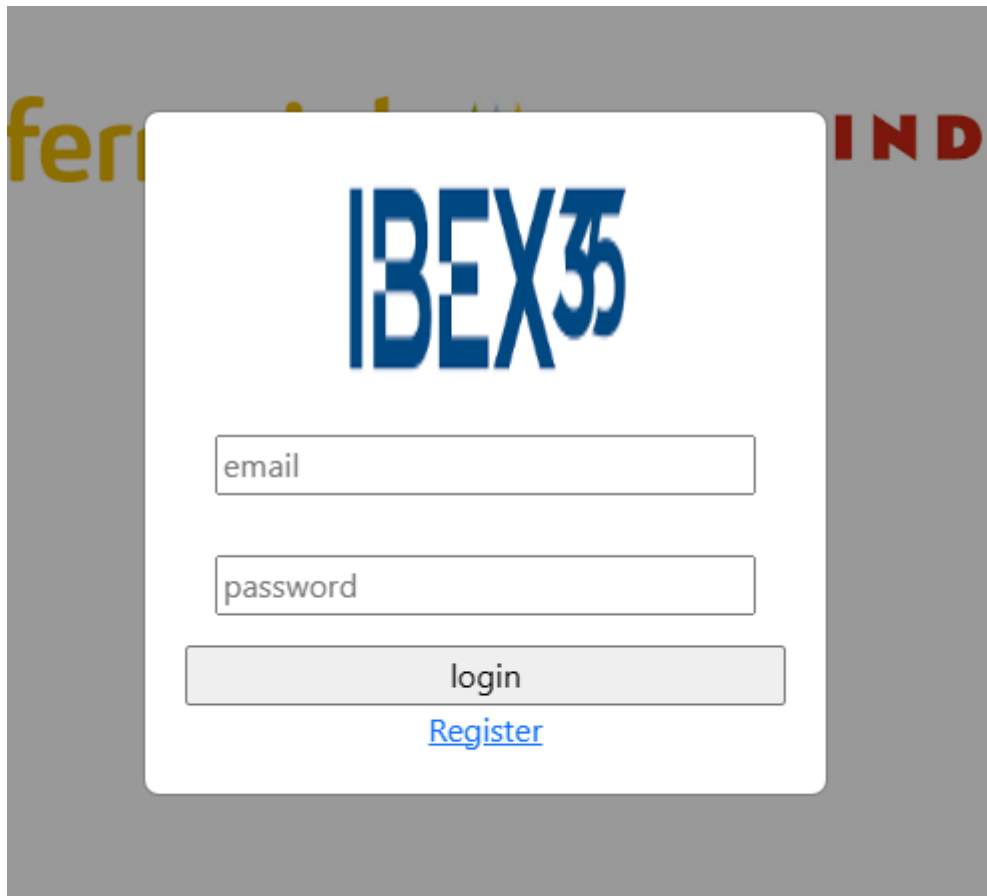
## Introducción

En esta documentación, se pretende mostrar una descripción más detallada del proyecto denominado IBEX 35. EL documento se estructura en diferentes apartados atendiendo a las asignaturas del curso: frontend ( DWC y DIW), api (DWS) , generador de datos y dockerizacion (DAW).

El proyecto trata de mostrar las cotizaciones en tiempo real de 10 empresas del IBEX 35. La aplicación consta de un SPA donde se podrán seleccionar las empresas que se quieren consultar. A continuación, se podrá ver la cotización real de la empresa y las gráficas que mostraran el histórico de datos.

# Frontend

El frontend se compone de una página SPA. Esto significa que se utilizara una sola página para mostrar toda la aplicación. Al iniciar la aplicación, se muestra un login/registro construido con un modal.



1. Login hecho con modal

No se podrá acceder a la aplicación hasta que se haya registrado o logueado el usuario.

- **DRAG & DROP**

Una vez dentro, aparecerá una drag and drop donde se podrán arrastrar las imágenes de las empresas al cajón de consulta. El drag and drop ha sido creado mediante la librería de jquery. Si queremos eliminar una empresa, la arrastramos a la papelera para devolverla al cajón de inicio.



## 2. Drag and drop

Si damos al botón guardar, se cargará la siguiente página y se ocultará la anterior. En la página siguiente, se mostrará las cotizaciones de las empresas seleccionadas.

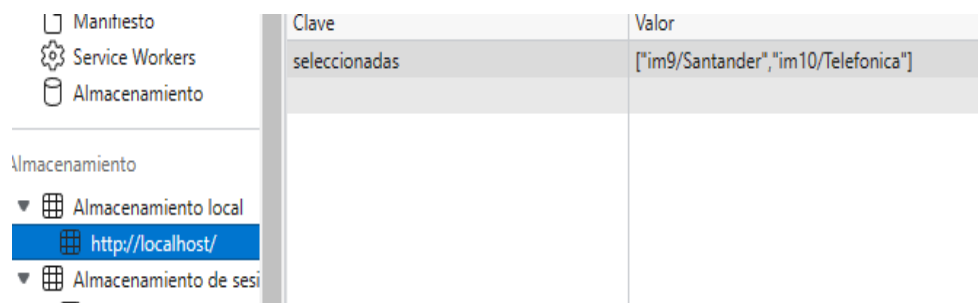


## 3. Actualización en tiempo real

Los valores se irán actualizando cada minuto y dependiendo de si sube o baja se pintara de verde o rojo respectivamente.

- **LocalStorage**

Cuando se hace el paso a la siguiente página, se guarda un localStorage que contiene una array con el id y nombre de la empresa que se ha seleccionado. Así, en el caso de que se salga de la aplicación, se volverá a entrar en la segunda página mostrando las empresas que se habían consultado con anterioridad.



The screenshot shows the Chrome DevTools interface. On the left, the 'Almacenamiento' (Storage) tab is selected, and 'Almacenamiento local' (Local Storage) is expanded. The URL 'http://localhost/' is highlighted. The main table displays the following data:

Clave	Valor
seleccionadas	["im9/Santander", "im10/Telefonica"]

4.Localstorage que guarda el id y nombre de la empresa

- **Llamadas Fetchs**

También, es importante añadir que tanto para el login/registro como la descarga de datos de las empresas se han realizado llamadas fetches a la api construida en Laravel.

En el caso del login y registro, se hace una llamada a la ruta `api/login` o `api/register` mediante un método POST. Al body, se pasarán los datos de email y contraseña insertados en los campos y como respuesta la api devolverá un token que se guardará en un `sessionStorage` necesario para poder realizar luego los fetchs de las empresas.

```

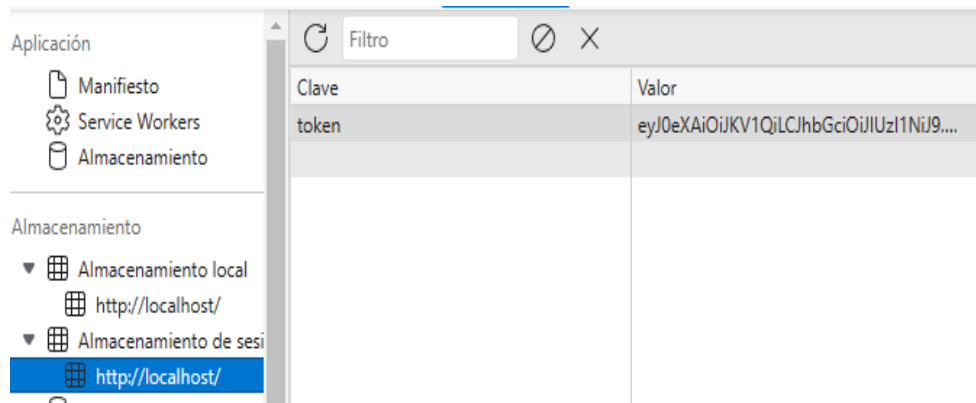
async function registrarUsuario(){
  const name = document.querySelector("#nameR");
  const email = document.querySelector("#emailR");
  const password = document.querySelector("#passR");

  try {
    const response = await fetch("http://localhost:80/api/register?name=", {
      method: 'POST',
      headers: {},
      body: new URLSearchParams({
        name: name.value,
        email: email.value,
        password: password.value
      })
    });

    if (response.ok) {
      const result = await response.json();
      setToken(result.authorisation.token);
      document.getElementById("myModal").style.display = "none";
    }
  } catch (err) {
    console.error(err);
  }
}

```

5. Ejemplo de fetch para registrar usuario



6. Sessionstorage que guarda el token.

En el caso de descargar los datos de la empresa, se harán fetches a la api de laravel mediante métodos GET. Para ello, en la variable opciones es necesario pasarle el token guardado en la sesión storage.

```

function consultarEmpresas(empresas,local){

  const options = {
    method: 'GET',
    headers: {
      Authorization: 'Bearer '+localStorage.token
    }
  };

  if(!local){
    fetch('http://localhost:80/api/empresas', options)
      .then(response => response.json())
      .then(response => prueba(response,empresas))
      .catch(err => console.error(err));
  }else{

    fetch('http://localhost:80/api/empresas', options)
      .then(response => response.json())
      .then(response => pruebaLocal(response))
      .catch(err => console.error(err));
  }

}

```

7.Ejemplo de fetchs para llamar a las empresas

Existen 2 fetches, uno que descarga las cotizaciones actuales de las 10 empresas y otro que pasándole un id como parámetro descarga todo su historial de datos. La primera es llamada cuando se muestran las cotizaciones de la empresa reales. La segunda cuando se desean consultar las gráficas.

Además, para las gráficas, se han utilizado diferentes funciones como “reduce” o “filter” para filtrar los datos.

```

const result = dataGuar.filter((obj, index, arr) => {
  const dateObj = new Date(obj.fecha);
  const nextDateObj = new Date(arr[index + 1] ? arr[index + 1].fecha : obj.fecha);
  return dateObj.getDate() !== nextDateObj.getDate() && dateObj.getHours() === 23 && dateObj.getMinutes() === 59;
});

```

8.Ejemplo de función filter

En este ejemplo, se utiliza el filter para filtrar los datos y que nos devuelva de cada día del mes su última cotización.

Para actualizar los datos, se ha implementado un setInterval que llamara a una función cada minuto para que realice un fetch. Esto se ha hecho así porque el generador de datos genera 10 datos para las 10 empresas cada minuto.

```
function empezarCiclo(){
    if(cicloConsulta != null){
        clearInterval(cicloConsulta);
    }
    cicloConsulta = setInterval(refrescarDatos,60000);
}
function refreshData(){
    // ...
}
```

9.Ciclo de consultas

## API

Para conectar el frontend con la base de datos se utiliza una api creada en Laravel. La api se utilizara en dos ocasiones: cuando se hace un login/registro en la página y cuando se quieran descargar datos de las empresas.

- **Login y Registro. Autenticación JWT**

Para el login/registro, se ha implementado un autenticación JWT(JSON Web Token). Con esta autenticación, cada vez que se registre o logee un usuario generará un token. Este token se utilizará para poder realizar las llamadas a las empresas.

Para ello, se ha creado un guard llamado 'api' que utiliza el driver jwt. De esta manera, podemos añadir un middleware tanto en el controlador de las autenticaciones(authcontroller) como el de las empresas (EmpresaController), para que solo puedan realizar aquellos métodos quien esté autenticado.

```
'guards' => [
    'web' => [
        'driver' => 'session',
        'provider' => 'users',
    ],

    'api' => [
        'driver' => 'jwt',
        'provider' => 'users',
    ],
],
```

10..Declaración del guard

```

3 references | 0 implementations
class AuthController extends Controller
{

    0 references | 0 overrides
    public function __construct()
    {
        $this->middleware('auth:api', ['except' => ['login','register']]);
    }

    0 references | 0 overrides
    public function login(Request $request)

```

11.Middleware para bloquear request.

Al modelo “User” existente se ha implementado la clase “JWTSubject” con los métodos “getJWTCustomClaims()” y “getJWTIdentifier()” para poder devolver el identificador JWT cuando este se haya generado.

```

5 references | 0 implementations
class User extends Authenticatable implements JWTSubject
{
    use HasFactory, Notifiable;

```

```

43      /**
44       * Get the identifier that will be stored in the subject claim of the JWT.
45       *
46       * @return mixed
47       */
48      0 references | 0 overrides
49      public function getJWTIdentifier()
50      {
51          return $this->getKey();
52      }
53      /**
54       * Return a key value array, containing any custom claims to be added to the JWT.
55       *
56       * @return array
57       */
58      0 references | 0 overrides
59      public function getJWTCustomClaims()
60      {
61          return [];
62      }
63  }

```

12.Funciones añadidas al modelo User



- **Controladores**

La api consta de dos controladores:

- AuthController: gestiona el login, registro y logout de la aplicación. El login y registro responderán con un json devolviendo el token generado, sino sacará un mensaje de error.

```
0 references | 0 overrides
public function login(Request $request)
{
    $request->validate([
        'email' => 'required|string|email',
        'password' => 'required|string',
    ]);
    $credentials = $request->only('email', 'password');

    $token = Auth::attempt($credentials);
    if (!$token) {
        return response()->json([
            'status' => 'error',
            'message' => 'Unauthorized',
        ], 401);
    }

    $user = Auth::user();
    return response()->json([
        'status' => 'success',
        'user' => $user,
        'authorisation' => [
            'token' => $token,
            'type' => 'bearer',
        ]
    ]);
}
```

13. Login del AuthController

```

0 references | 0 overrides
public function register(Request $request){
    $request->validate([
        'name' => 'required|string|max:255',
        'email' => 'required|string|email|max:255|unique:users',
        'password' => 'required|string|min:6',
    ]);

    $user = User::create([
        'name' => $request->name,
        'email' => $request->email,
        'password' => Hash::make($request->password),
    ]);

    $token = Auth::login($user);
    return response()->json([
        'status' => 'success',
        'message' => 'User created successfully',
        'user' => $user,
        'authorisation' => [
            'token' => $token,
            'type' => 'bearer',
        ]
    ]);
}

```

#### 14. Registro del AuthController

- EmpresaController: En este se utilizan dos funciones. Un método llamado index que devuelve las cotizaciones actuales y otra llamada show que pasándole un id devuelve el histórico de datos de esa empresa.

```

1 reference | 0 overrides
public function index(Request $request)
{
    $empresas = Actual::all();
    return $empresas;
}

```

```

1 reference | 0 overrides
public function show($id)
{
    $historico = new Historico;
    $empresa = $historico->datos($id);
    return $empresa;
}

```

## ● Rutas api

Las rutas para hacer las llamadas se han definido en la carpeta Routes/api.

```
Route::middleware('auth:sanctum')->get('/user', function (Request $request) {  
    return $request->user();  
});  
  
Route::get('/empresas', [EmpresaController::class,'index']); //mostrar todas la cotizaciones  
Route::get('/empresas/{id}', [EmpresaController::class,'show']); //muestra una cotizacion  
  
Route::controller(AuthController::class)->group(function () {  
    Route::post('login', 'login');  
    Route::post('register', 'register');  
    Route::post('logout', 'logout');  
    Route::post('refresh', 'refresh');  
});
```

15.Rutas api

Para las llamadas a las empresas, se utilizan 2 rutas tipo GET:

- 'api/empresas': esta ruta llama a la función 'index' del controlador 'EmpresaController' que devuelve todo el los valores de las cotizaciones actuales de las 10 empresa desde la tabla 'actuales'.
- 'api/empresas/{id}': este llama al método 'show' que pasándole el id de una empresa como parámetro devuelve el histórico de esa empresa.

Para las rutas del "authcontroller", existen 3 rutas tipo post (login, register y logout) que llaman a sus respectivos métodos del AuthController.

## ● Modelos

Por último, se han creado tres modelos que hacen referencia a las tres tablas que existen en la base de datos: "Actual" para la tabla "actuales" que contiene las cotizaciones actualizadas de las 10 empresas, "Empresas" para la tabla "empresas" que define las empresas y, "Histórico" para la tabla "historial\_empresas" donde se registra todo el histórico de datos.

En el caso del Modelo del "historial\_empresas" se ha creado una función que. pasándole un id como parámetro, descargue todo el historial de esa empresa ordenada de manera ascendente.

```

2 references
protected $table = 'historial_empresas';

0 references | 0 overrides
public static function datos($id) {
    return self::where('id_empresa', $id)->orderBy('fecha', 'asc')->get();
}
}

```

16.Función historial de datos

## Docker

La aplicación está enteramente dockerizada. Antes de explicar como se ha dockerizado, es necesario ver cómo se estructura esta.

La aplicación consta de un front desde donde el usuario podrá acceder para hacer sus consultas. Este front está dockerizado en un nginx mediante un dockerfile.

```

# Use an image de nginx como base
FROM nginx

# Copiar los archivos a la carpeta de trabajo de nginx
COPY . /usr/share/nginx/html

# Establecer la carpeta de trabajo
WORKDIR /usr/share/nginx/html

# Exponer el puerto 80
EXPOSE 80
# RUN npm install chart.js

# Ejecutar nginx
CMD ["nginx", "-g", "daemon off;"]

```

17.Dockerfile del front

Cuando se hagan llamadas a la aplicación, se harán a través de una api de laravel. Esta api está contenerizada en un apache. Además, para este proyecto, se han utilizado 2 apis que son balanceadas por un nginx que hace de balanceador de carga. De esta manera, el nginx irá desviando las peticiones a una api o la otra dependiendo de si una está ocupada o no.

```

events {}
http {
    upstream web{
        server front:80;
    }
    upstream api {
        server laravel1:80;
        server laravel2:80;
    }

    server {
        listen 80;

        location / {
            proxy_pass http://web;
        }
        location /api {
            proxy_pass http://api;
        }
    }
}

```

#### 18. Balanceador de carga

La base de datos está contenerizada en un mysql. Al levantar el docker, la base de datos ejecuta un sql que se encuentra en una carpeta “dump” donde se crearán todas las tablas necesarias para dejar la aplicación inicializada. Las tablas son: una denominada “actuales” donde se subirán las últimas cotizaciones de cada empresa, otra “historial\_empresas” con todo el histórico de datos de cada empresa y una última llamada “empresas” con la definición de las empresas.

Por último, se encuentra el generador de datos. Este será ejecutado en un node dockerizado una vez que la base de datos haya sido desplegada. Mediante un script, primero generará un histórico de datos de las 10 empresas durante el último mes con un dato por minuto. Después, a través de la librería chron js, generará un dato por minuto y los irá añadiendo a la base de datos.

```

# Use an image de Node.js como base
FROM node:14

# Establecer la carpeta de trabajo
WORKDIR /app

# Copiar los archivos a la carpeta de trabajo
COPY . .

# Instalar las dependencias
RUN npm install

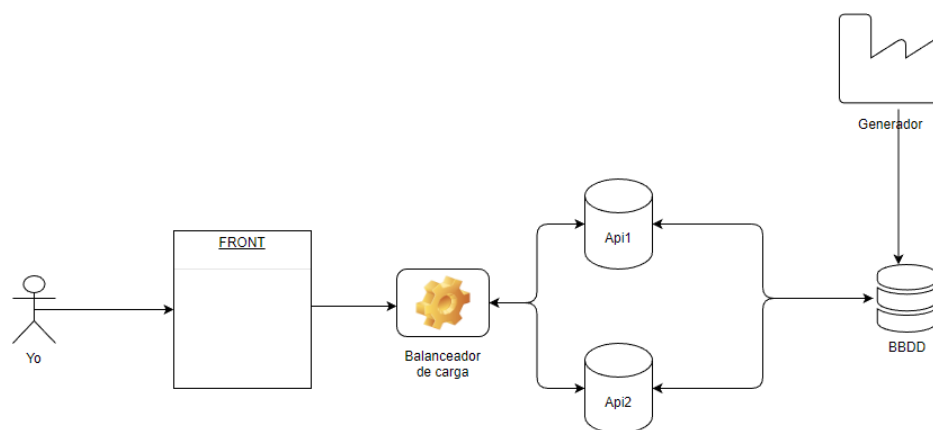
# Exponer el puerto predeterminado para la aplicación
EXPOSE 3000
# EXPOSE 3306

# Ejecutar la aplicación
# CMD ["npm", "start"]
CMD [ "node","generate-data.js" ]

```

19.Dockerfile del generador de datos

Con todo esto, el esquema de la aplicación quedaría como el reflejado en la siguiente gráfica:



20.Grafica de dockerizacion del proyecto

- **Docker Compose**

El docker compose es el siguiente:

```
version: '3.7'

services:
  front:
    build: ./Front
    depends_on:
      - db
    volumes:
      - ./Front:/usr/share/nginx/html
    networks:
      - MarkOSnet

  generador:
    build: ./GeneradorDatos
    # ports:
    #   - 3000:80
    volumes:
      - ./GeneradorDatos:/app
    depends_on:
      db:
        condition: service_healthy
    networks:
      - MarkOSnet

  laravel1:
    build: ./api
    # command: php artisan migrate:fresh --seed
    image: laravel.prod
    volumes:
      - ./api:/src/app
    # ports:
    #   - 8000:80
    depends_on:
      - db
    networks:
      - MarkOSnet

  laravel2:
    build: ./api
    # command: php artisan migrate:fresh --seed
    image: laravel.prod
    volumes:
      - ./api:/src/app
    # ports:
    #   - 8000:80
    depends_on:
      - db
    networks:
```

```

db:
  image: mysql
  command: --max_allowed_packet=325058560
  environment:
    MYSQL_ROOT_PASSWORD: stocks
    MYSQL_USER: stocks
    MYSQL_DATABASE: stocks
    MYSQL_PASSWORD: stocks
  volumes:
    - ./dump:/docker-entrypoint-initdb.d
    - mysql:/var/lib/mysql
  ports:
    - 3306:3306
  networks:
    - MarkOSnet
  healthcheck:
    test: ["CMD", "mysqladmin", "ping", "-h", "localhost"]
    timeout: 20s
    retries: 10

phpmyadmin:
  depends_on:
    - db
  image: 'phpmyadmin/phpmyadmin:latest'
  restart: always
  environment:
    PMA_HOST: db
    PMA_PORT: 3306
    PMA_USER: 'stocks'
    PMA_PASSWORD: 'stocks'
  networks:
    - MarkOSnet
  ports:
    - 83:80

proxy:
  image: nginx
  ports:
    - "80:80"
  volumes:
    - ./nginx/nginx.conf:/etc/nginx/nginx.conf:ro
  networks:
    - MarkOSnet
  depends_on:
    - laravel1
    - laravel2

networks:
  MarkOSnet:
    driver: bridge

volumes:

```

20. Docker compose

Como aporte curioso que añadir:



- Al depend\_on del generador le añadió una condición de service\_healthy. De manera que no comenzará hasta que la base de datos haya pasado una prueba de healthcheck.
- Construir un volumen a la base de datos. Así se puede borrar cuando elimines el docker y evitas la persistencia de datos.

```
volumes:  
  mysql:  
    driver: local
```

## Generador de Datos

El generador de datos se compone de tres fases:

```
1  
2  try {  
3    // await createDb();  
4    // await insertCompanies()  
5    await insertCompaniesData(ENTRIES_PER_COMPANY);  
6    await inicializarDatos();  
7    cron.schedule("* * * * *", () =>{  
8      insertCompaniesData(1);  
9      console.log("insertado a fecha: " +new Date());  
10   });  
11 } catch (e) {  
12   console.error(e)  
13 }  
14
```

21. generador

- Histórico: primero ejecuta un histórico datos del último mes. Para ello, ejecuta un función que crea un dato aleatorio en base al último dato aleatorio creado. Esto se hace cada minuto.

```
const f = (old_price, volatility) => {
  const rnd = Math.random() - 0.498;
  const change_percent = 2 * volatility * rnd;
  const change_amount = old_price * change_percent;
  const new_price = old_price + change_amount;

  if (new_price < 0.01) return new_price + Math.abs(change_amount) * 2;
  else if (new_price > 1000) return new_price - Math.abs(change_amount) * 2;

  return new_price;
};
```

22. función aleatoria

- Inicialización: Después ejecuta una función para crear unos datos actualizados a día de hoy. Este dato se inserta en el historial de la empresa y se actualiza en la tabla de “actuales”.

```
const inicializarDatos = async () =>{
  try {
    const con = await pool.getConnection();
    const query1 = `SELECT * FROM historial_empresas ORDER BY fecha DESC LIMIT 10`;
    const query2 = `update actuales set datos = ?, fecha = ? where id = ?`;
    const [rows] = await con.query(query1);
    console.log(rows)
    rows.forEach(async ele=>{
      await con.query(query2,[ele.valor,ele.fecha,ele.id_empresa]);
      console.log("insertado en fila: "+ele.id_empresa);
    })

    con.release();
  } catch (e) {
    console.error(e)
  }
}
```

23. inicialización de datos

- Bucle: Por último, mediante la librería de node cron.js, generamos un bucle que cada minuto va generando un nuevo dato en base al último valor creado, para que no se produzca mucha dispersión.

```
cron.schedule("* * * * *", () =>{
  insertCompaniesData(1);
  console.log("insertado a fecha: " +new Date());
});
```


24.bucle con cron

## Interfaz Web

La interfaz web se compone de una página SPA. Esto significa que todo el contenido se irá desplegando y ocultando en la misma página.

- **Login Modal**

Nada más acceder a la página, se desplegará un login modal desde el que habrá que registrarse o loguearse para poder continuar.



25. Login realizado con modal

- **Página principal**

Una vez logueado, se desplegará una página con una cabecera que contiene el título y un monigote para poder realizar un logout. Debajo aparece el drag and drop necesario para seleccionar las empresas que se quieren consultar.

Cuando se pulse el botón guardar, se ocultara la página y se mostrará la siguiente con las empresas seleccionadas para consultar.

Esta página es responsive de manera que se adapta a cualquier tamaño de pantalla.



26. Página principal responsive

- Página consultas

La información de estas empresas se mostrará con un card implementado con bootstrap.

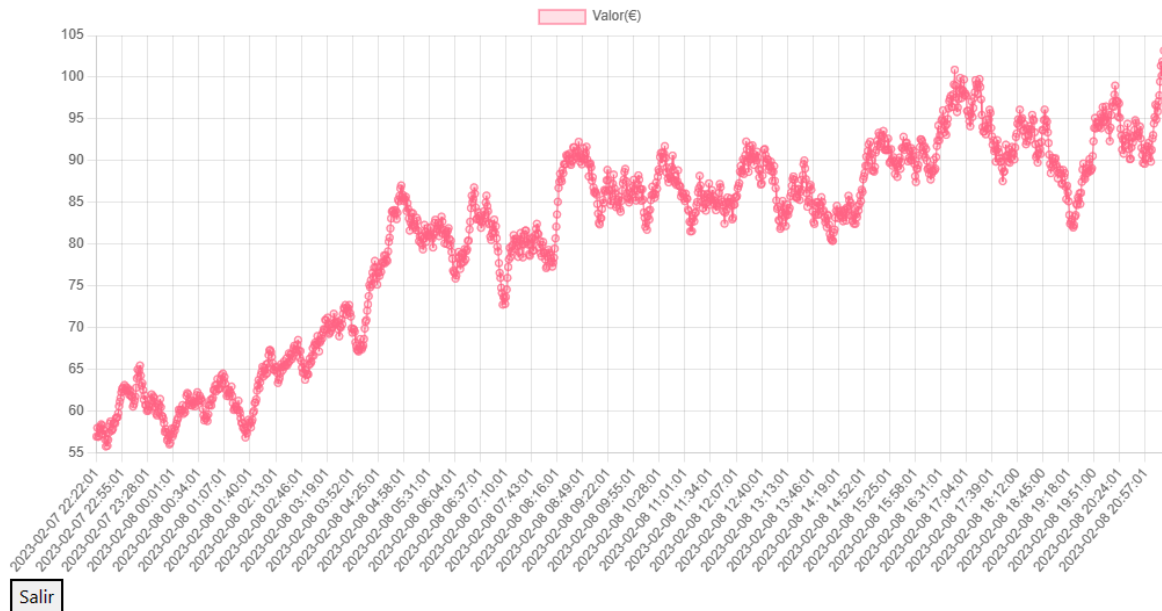


#### 27. Página principal responsive

Dentro, aparecerá la cotización actual y un botón para mostrar las gráficas de la empresa.

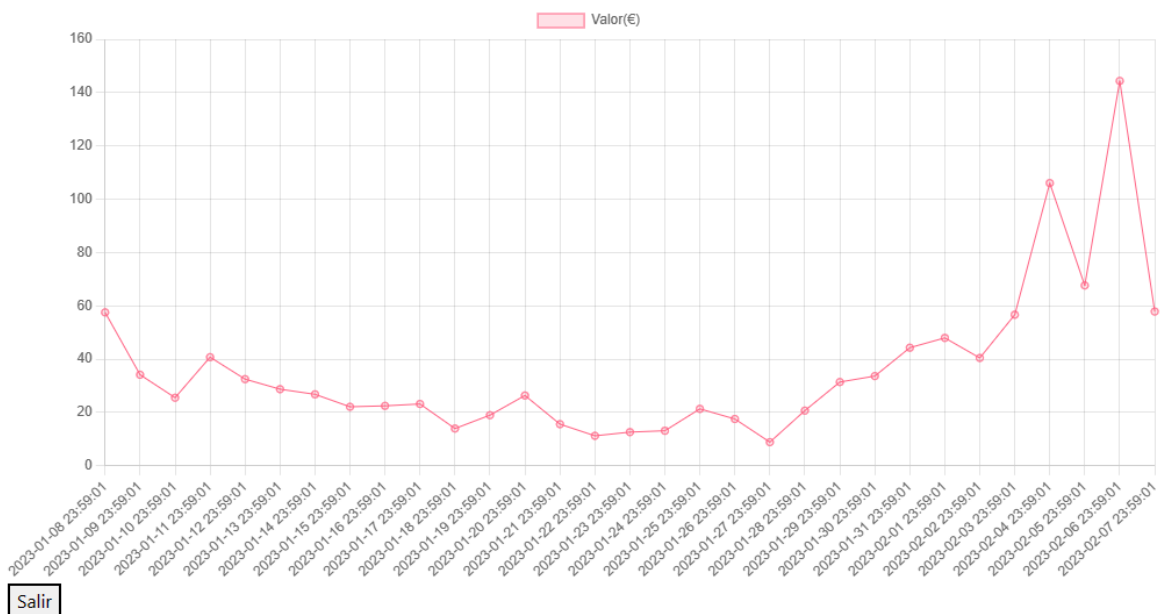
Las gráficas se han realizado con la librería chart.js. Cuando se pulsa en el botón “Gráfico”, se despliega un chart que muestra las cotizaciones de la empresa durante las últimas 24 horas. Existe la posibilidad de elegir si se quiere mostrar también las cotizaciones de cada día durante el último mes.

Mes 24H



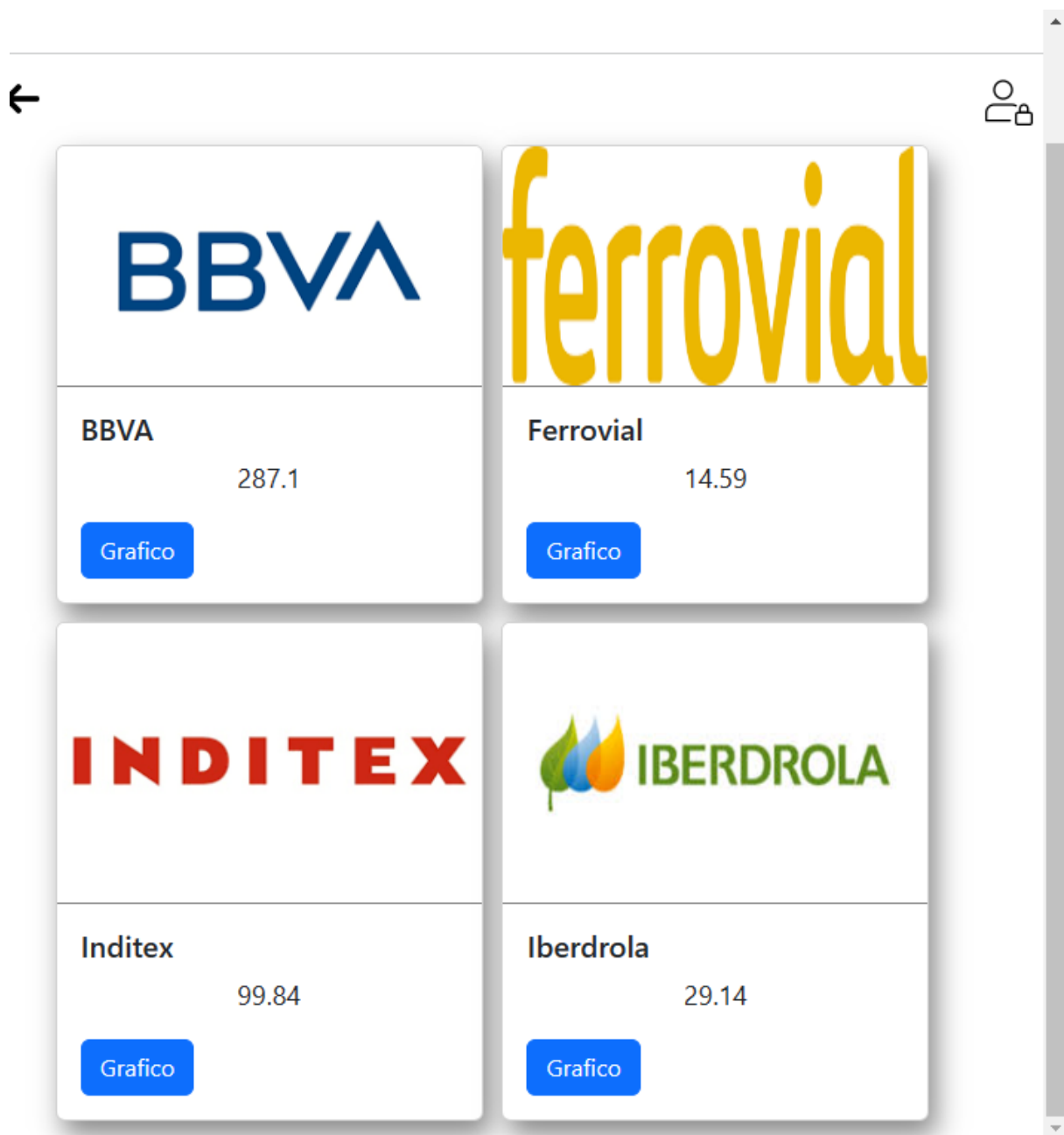
27. Gráfico con los valores de cada minuto de las últimas 24 horas

Mes 24H



28. Gráfica que muestra las cotizaciones durante un mes

Además, la aplicación es responsive, de manera que se adapta a cualquier tipo de tamaño, y es accesible a cualquier tipo de usuario.



29. Ejemplo de tamaño responsive