

MANUAL DEL PROGRAMADOR

Ciente: TAP

Proyecto: Landing CampViral



Versión: 1.2.0

Autor: Equipo 1 - 2B BE - **Carlos Suárez, Diana Fernández, Nicolás Zapata, Wei Lee**

Contenido

OBJETIVO DEL MANUAL	4
SOBRE LA LANDING WEB	4
TECNOLOGÍAS UTILIZADAS	4
STRAPI	5
REACT	5
POSTGRESQL	6
INTRODUCCIÓN A STRAPI	6
OBTENCIÓN DEL CÓDIGO Y REQUISITOS	6
VARIABLES DE ENTORNO	7
AUTENTICACIÓN CONTRA EL SERVIDOR	8
ENTIDADES	10
DIAGRAMA DE ENTIDAD RELACIÓN	18
DIAGRAMA DE SECUENCIA	19
PAQUETES ADICIONALES INSTALADOS	21
PARTICIPANTS	21
PROMOTIONS	28
COUPONS	30
BASE DE DATOS	31
INTRODUCCIÓN A REACT	34
VARIABLES DE ENTORNO	34
DISTRIBUCIÓN DE CARPETAS	35
COMPONENTES	40
FORMULARY	40
RANKING	50
COUPONMESSAGE	62

TERMSMESSAGE	68
APP	73
UTILS	87

Objetivo del Manual

En el presente manual, se describe la estructura del proyecto web y la respectiva implementación de sus componentes. Tiene por objetivo lograr que cualquier programador con solo leerlo pueda entender el flujo del proyecto para poder actuar sin precedentes sobre todas sus partes.

Sobre la Landing Web

La presente Landing Web está enfocada en la viralización de campañas, donde los usuarios mediante un formulario de registro pueden inscribirse para obtener promociones ofrecidas por TAP. Además, la misma, ofrece la posibilidad de participar por premios a través de un ranking donde el participante pueda ir sumando puntos compartiendo, si lo desea, un link para que demás personas se sumen.

Tecnologías utilizadas

Las siguientes tecnologías fueron utilizadas a lo largo de todo el proyecto. Eso incluye tanto para el frontend como el backend:

Strapi

Strapi es un headless CMS de código abierto desarrollado en Node.js y con un enfoque en lograr una gran capacidad de personalización, haciendo posible proyectos complejos en un periodo de tiempo más reducido.

Como casi todo headless CMS, tiene la capacidad de generar tipos de entradas repetidos (collections types) en base a nuestras necesidades así como también contenidos únicos (single types), y obtenerlos a través de su API REST o GraphQL. Además de brindar herramientas para autenticación manual y con redes sociales, envío de emails, subida de archivos, gestionar roles y permisos de usuarios, configurar webhooks y más.

React

ReactJS es una librería JavaScript de código abierto enfocada a la visualización. Esta tecnología nos permite el desarrollo de interfaces de usuario de forma sencilla, lo cual es posible gracias a los componentes interactivos y reutilizables que la misma nos permite crear.

React está basado en un paradigma llamado programación orientada a componentes en el que cada componente es una pieza con la que el usuario puede interactuar. Estas piezas se crean usando una sintaxis llamada JSX permitiendo escribir HTML (y opcionalmente CSS) dentro de objetos JavaScript.

Estos componentes son reutilizables y se combinan para crear componentes mayores hasta configurar una web completa.

Esta es la forma de tener HTML con toda la funcionalidad de JavaScript y el estilo gráfico de CSS centralizado y listo para ser abstraído y usado en cualquier otro proyecto.

PostgreSQL

PostgreSQL es un sistema de código abierto de administración de bases de datos del tipo relacional. En este sistema, las consultas relacionales se basan en SQL.

Una de las herramientas de suma utilidad que nos incluye PostgreSQL es la llamada 'pgAdmin'. Esta herramienta nos permite desde hacer búsquedas SQL hasta desarrollar toda nuestra base de datos de forma muy fácil e intuitiva; directamente desde la interfaz gráfica.

Introducción a Strapi

Como ya hemos mencionado anteriormente, Strapi es un sistema de gestión de contenidos o CMS (del inglés content management system) de código abierto y 100% Javascript, que permite crear un entorno de trabajo para la creación y administración de contenidos mediante API REST. Sirve tanto para personalizar y cambiar el contenido de la página WEB sin tener que modificar el código, como para agregar la lógica del backend y realizar ABM en la base de datos.

Obtención del código y requisitos

Es necesario contar con node.js y npm instalado, clonar el repositorio https://github.com/nicozapata-ort/tap_web_ort_strapi y ejecutar por consola el comando NPM INSTALL para descargar todos los paquetes y dependencias.

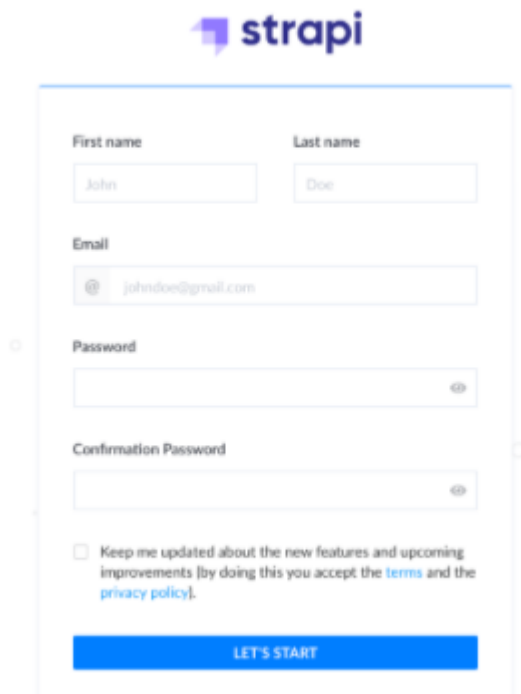
Variables de entorno

Antes de poder ejecutar el backend se deben tener definidas las siguientes variables de entorno, en el proyecto se incluye el archivo “.env.example” con las variables necesarias, se recomienda copiarlo o renombrarlo a “.env” y agregar los valores necesarios:

- DATABASE_HOST: IP o URL de donde este hosteada la base de datos.
- DATABASE_PORT: puerto a utilizar para conectarse a la base de datos.
- DATABASE_USERNAME: usuario que será utilizado para acceder a la base de datos.
- DATABASE_PASSWORD: contraseña del usuario anterior definido para acceder a la base de datos.
- DATABASE_SCHEMA: string utilizado para el esquema de la base de datos.
- DATABASE_NAME: nombre de la base de datos.
- URL_LANDING: IP o URL donde esté hosteada la landing, incluyendo de ser necesario el puerto.
- HOST: IP o URL donde estará hosteado el servidor.
- PORT: Puerto que será utilizado para el servidor.
- ADMIN_JWT_SECRET: string alfanumérico que será utilizado para codificar los tokens JWT.

Autenticación contra el servidor

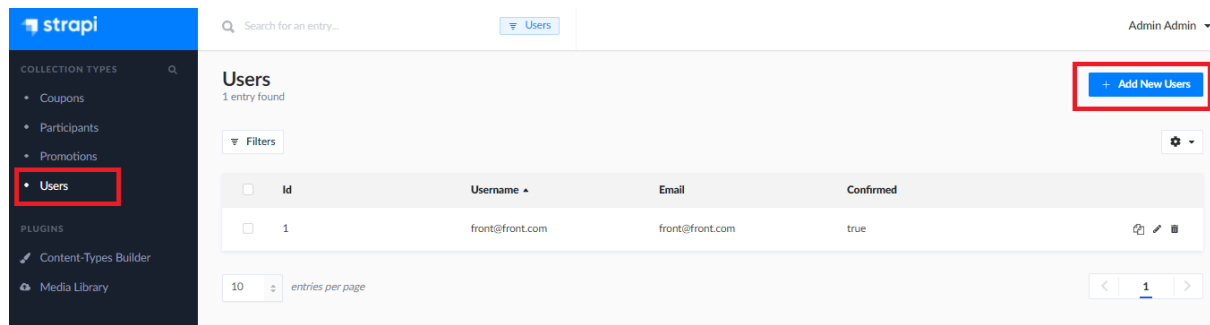
La primera vez que se ejecute el backend, al acceder a la ruta de administrador (/admin) se pedirá crear el usuario administrador, se deberán completar los datos de nombre, apellido, email y luego ingresar y confirmar una contraseña:



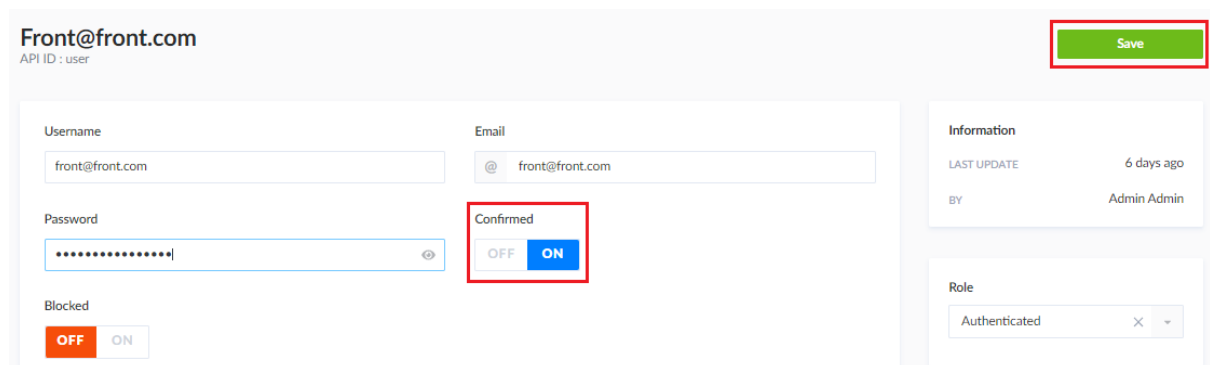
The image shows the Strapi admin interface for creating a new user. At the top, the Strapi logo is visible. Below it, there is a form with the following fields:

- First name:** A text input field with the value "John".
- Last name:** A text input field with the value "Doe".
- Email:** A text input field with the value "john.doe@gmail.com".
- Password:** A text input field with a password strength indicator (two eyes icon).
- Confirmation Password:** A text input field with a password strength indicator (two eyes icon).
- ☐ **Keep me updated about the new features and upcoming improvements** (by doing this you accept the [terms](#) and the [privacy policy](#)).
- LET'S START** button.

Una vez ingresado se verá una pantalla como la siguiente y se deberá crear un usuario, el cual luego será usado en el front-end para poder autenticarse contra el servidor, en la colección Users:



Una vez creado el usuario se deberá confirmar que el usuario quede como confirmado “ON”, luego guardarlo:



Una vez se tenga el email y la contraseña registrada en el backend, para obtener el token de autorización del usuario para el front-end hay dos maneras, mediante código o mediante una aplicación que permita realizar peticiones HTTP:

1. Mediante código se debe utilizar una librería que permita realizar peticiones, en este ejemplo se utiliza axios, se debe reemplazar <http://localhost:1337> por la url base donde se esté ejecutando el backend, se debe mantener la ruta /auth/local, y se debe reemplazar en el atributo identifier el email que se haya registrado, y en password la contraseña con la que se registró dicho email:

```
import axios from 'axios'
const { data } = await axios.post('http://localhost:1337/auth/local', {
  identifier: 'example@example.com',
  password: 'Password123',
});
console.log(data.jwt);
```

El console log imprimirá el token necesario para poder realizar peticiones.

2. Mediante una aplicación que permita realizar peticiones HTTP, se deberá realizar una petición con método POST a la url base, en la ruta auth/local, y se debe adjuntar en el body de la petición un JSON como el que se muestra a continuación, se debe reemplazar en el atributo identifier el email que se haya registrado, y en password la contraseña con la que se registró dicho email:

```
{
  identifier: 'example@example.com',
  password: 'Password123',
}
```

Entidades

La colección **Users** hace referencia a los usuarios que se pueden tener registrados y que pueden o no tener permiso sobre las promociones, cupones o los participantes.

- Esta colección no es modificable mediante código ya que viene precargada en Strapi.

La colección **Promotions** es donde se deben cargar las promociones de las campañas, cuenta con los siguientes campos:

- **Description:** hace referencia a la descripción de la campaña:
- **dateMin:** Atributo de tipo fecha que indica la fecha de inicio de la campaña.
- **dateMax:** Atributo de tipo fecha que indica el final de la campaña.
- **prizeMinPrice:** Atributo que hace referencia al premio que obtendrían
- **prizeMaxPrice:** atributo numérico que hace referencia al premio del ranking que obtendrían desde la posición 1 hasta el atributo **firstPlaces**.
- **maxParticipants:** hace referencia a la cantidad de participantes que se mostrarán en el ranking cuando un usuario lo consulte.
- **couponPrizeLongDescription:** hace referencia a la descripción del premio obtenido.
- **couponPrizeShortDescription:** hace referencia a la descripción de la condición del premio obtenido.
- **descriptionSharePrizeCoupon:** hace referencia a la descripción de la posibilidad de compartir un link para sumar puntos.
- **firstPlaces:** referencia a la cantidad de usuarios que obtendrán el **prizeMaxPrice**.
- **descriptionFirstPrize:** atributo que hace referencia a la descripción del **prizeMaxPrice** (primer premio).
- **descriptionSecondPrize:** atributo que hace referencia a la descripción del **prizeMinPrice** (segundo premio).
- **promotionExpiredMessage:** mensaje a mostrar en la página cuando la campaña haya expirado.

- noAvailableCouponsMessage: mensaje a mostrar en la página si no hay cupones disponibles para esa campaña.
- coupons: lista de cupones asociados a esa campaña.
- participants: lista de participantes que se han inscrito en la campaña.

Las modificaciones de estos atributos mediante código se pueden realizar en la ruta: `api/promotion/models/promotion.settings.json` , compuesto por el siguiente código:

```
{
  "kind": "collectionType",
  "collectionName": "promotions",
  "info": {
    "name": "Promotion",
    "description": ""
  },
  "options": {
    "increments": true,
    "timestamps": true,
    "draftAndPublish": true
  },
  "pluginOptions": {},
  "attributes": {
    "title": {
      "type": "string",
      "maxLength": 20,
      "required": true
    },
    "dateMin": {
      "type": "date",
      "required": true
    },
    "dateMax": {
      "type": "date",
      "required": true
    },
    "description": {
      "type": "richtext",
      "required": true
    },
    "prizeMinPrice": {
      "type": "float",
      "required": true
    }
  }
}
```

```
"prizeMaxPrice": {
  "type": "float",
  "required": true
},
"maxParticipants": {
  "type": "integer",
  "required": true
},
"couponPrizeLongDescription": {
  "type": "text",
  "maxLength": 100,
  "required": true
},
"couponPrizeShortDescription": {
  "type": "string",
  "maxLength": 70,
  "required": true
},
"descriptionSharePrizeCoupon": {
  "type": "text",
  "maxLength": 80,
  "required": true
},
"firstPlaces": {
  "type": "integer",
  "required": true
},
"descriptionFirstPrize": {
  "type": "text",
  "maxLength": 30,
  "required": true
},
"descriptionSecondPrize": {
  "type": "text",
  "maxLength": 30,
  "required": true
},
"promotionExpiredMessage": {
  "type": "richtext",
  "required": true
},
"noAvailableCouponsMessage": {
  "type": "richtext",
  "required": true
},
"Picture": {
  "model": "file",
  "via": "related",
  "allowedTypes": [
    "images",
    "files",
    "videos"
  ]
}
```

```
    ],
    "plugin": "upload",
    "required": true,
    "pluginOptions": {}
  },
  "coupons": {
    "via": "promotion",
    "collection": "coupons"
  },
  "participants": {
    "collection": "participants",
    "via": "promotion"
  },
  "descriptionTermsAndConditions": {
    "type": "text"
  }
}
```

Colección **Coupons** son los cupones que pertenecen a cada promoción y reciben los usuarios para obtener el beneficio de la promoción, están compuestos por:

- coupon: representa la cadena alfanumérica que recibe el usuario por registrarse.
- used: tiene dos posibles valores, falso, si el cupón no ha sido enviado y está disponible para enviar a algún cliente que se esté registrado, y verdadero si el cupón ya ha sido enviado a algún cliente que ya se registró.
- promotion: id de la promoción a la cual está asociado el cupón.

Las modificaciones de estos atributos mediante código se deben realizar en la ruta: `api/coupons/models/coupons.settings.json` , y el archivo está compuesto por el siguiente código:

```
{
  "kind": "collectionType",
  "collectionName": "coupons",
  "info": {
    "name": "coupons",
    "description": ""
  },
  "options": {
    "increments": true,
    "timestamps": true,
    "draftAndPublish": true
  },
}
```

```
  "pluginOptions": {},
  "attributes": {
    "coupon": {
      "type": "string",
      "required": true
    },
    "used": {
      "default": false,
      "type": "boolean",
      "required": true
    },
    "promotion": {
      "via": "coupons",
      "model": "promotion",
      "required": true
    }
  }
}
```

Colección **Participants** son los clientes o usuarios que se registran en la lading para recibir el beneficio, tiene los siguientes atributos:

- name: nombre del cliente.
- lastname: apellido del cliente.
- dni: documento con el que se registra el cliente.
- phone: teléfono del cliente.
- referrals: cantidad de usuarios que se han registrado utilizando el link que se le genera al usuario.
- token: código único que se le genera al usuario para generar el link para referir usuarios y ganar puntos en el ranking.
- email: correo electrónico con el que se registra el cliente.
- normalizedEmail: correo electrónico del usuario todo en mayúscula para poder comparar y evitar duplicados.
- lastReferral: hora y fecha en la cual se sumó el último punto en el ranking, este valor sirve para desempatar en caso de que más de dos o más usuarios tengan el mismo puntaje, aquel que haya alcanzado primero el puntaje es el que se posiciona primero en el ranking.
- promotion: id de la promoción donde se registró el usuario, de esta manera si hay más de una campaña activa los usuarios se podrían registrar en cualquier y se genera un registro distinto por cada campaña para poder llevar una cuenta separada de cada puntaje.

Las modificaciones de estos atributos mediante código se deben realizar en la ruta: `api/participants/models/participants.settings.json` , que está compuesto por el siguiente código:

```
{  
  "kind": "collectionType",  
  "collectionName": "participants",  
  "info": {
```



```
    "name": "Participants",
    "description": ""
  },
  "options": {
    "increments": true,
    "timestamps": true,
    "draftAndPublish": true
  },
  "pluginOptions": {},
  "attributes": {
    "name": {
      "type": "string"
    },
    "lastname": {
      "type": "string"
    },
    "dni": {
      "type": "integer"
    },
    "phone": {
      "type": "integer"
    },
    "referrals": {
      "type": "integer"
    },
    "token": {
      "type": "uid"
    },
    "email": {
      "type": "email"
    },
    "normalizedEmail": {
      "type": "email"
    },
    "lastReferral": {
      "type": "datetime"
    },
    "promotion": {
      "via": "participants",
      "model": "promotion"
    }
  }
}
```

Diagrama de entidad relación

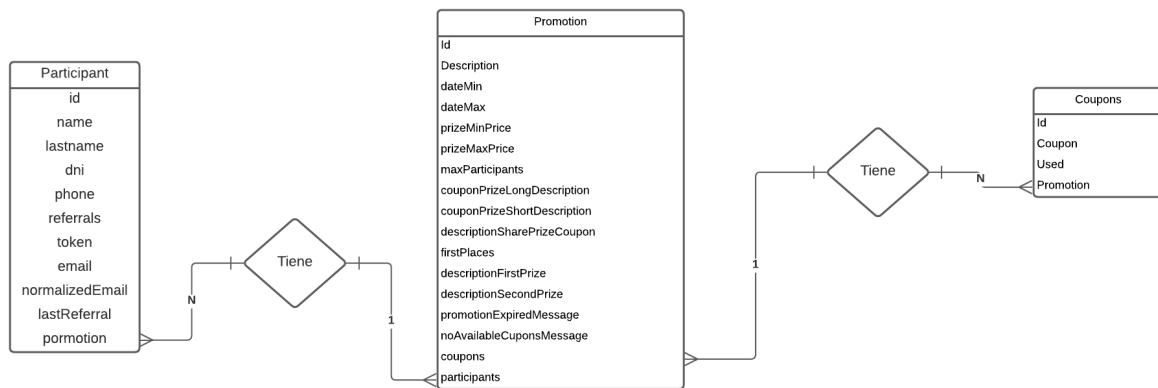
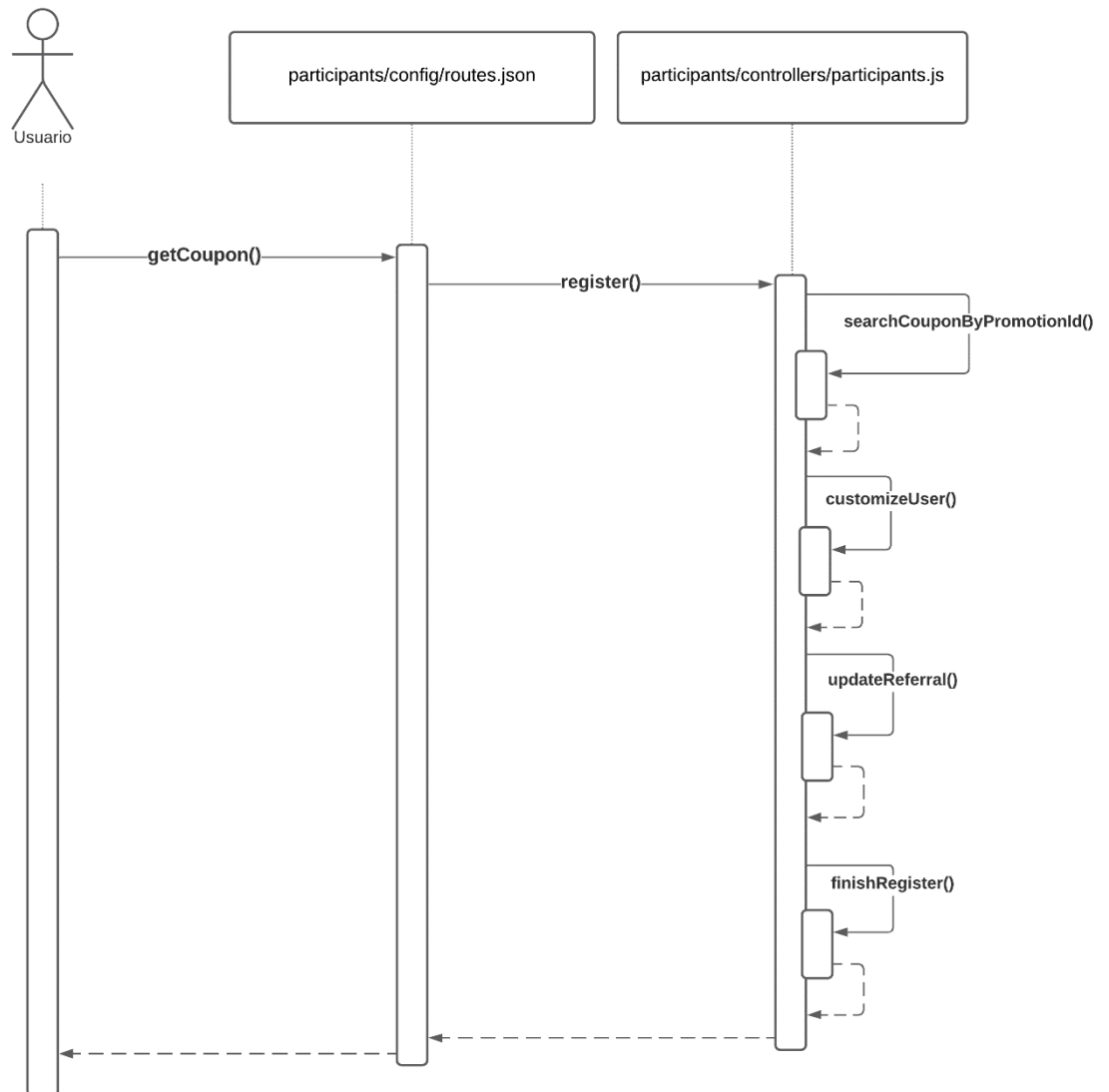
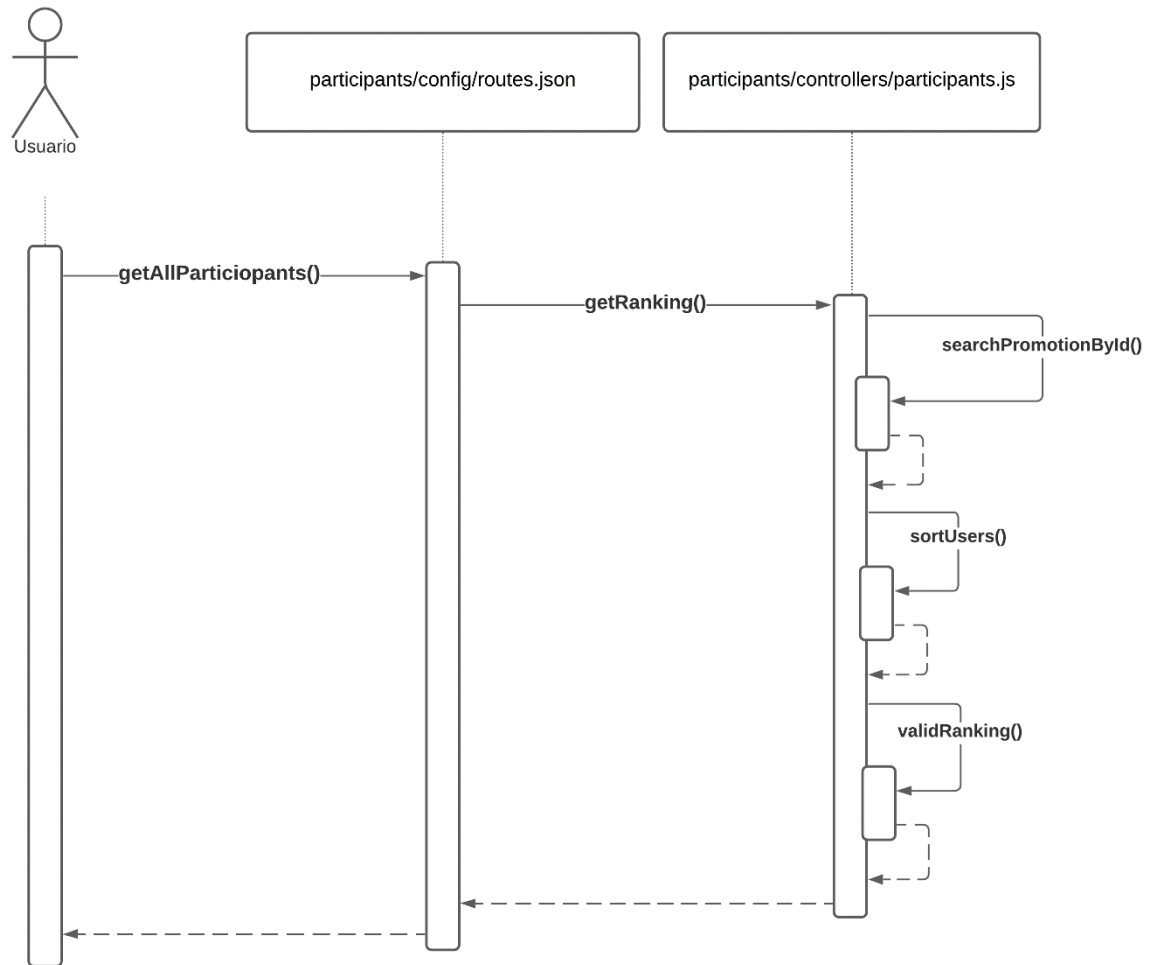


Diagrama de secuencia

- Registro de usuario:



- **Obtener ranking:**



Paquetes adicionales instalados

-uuid: paquete para generar el Token del usuario que se usará para generar el link único. Documentación: <https://www.npmjs.com/package/uuid>

La lógica creada para el correcto funcionamiento de lo pedido por el cliente se encuentra en la carpeta api.

Participants

Rutas

Ruta	Método	Función
/participants	GET	find
/participants/count	GET	count
/participants/:id	GET	findOne
/participants	POST	register
/ranking	GET	getRanking
/participants/:id	PUT	update
/participants/:id	DELETE	delete

- **POST: /participants**

Para poder hacer una petición a esta ruta se debe contar con un código de autenticación y se debería recibir un objeto con las siguientes características:

- name: string con una longitud mayor o igual a dos caracteres, hace referencia al nombre del cliente.
- lastname: string de una longitud mayor o igual a dos caracteres, hace referencia al apellido del cliente.
- dni: valor numérico entre 100000 y 99999999, hace referencia al documento de identidad del cliente.
- email: string que cumpla con el siguiente formato [xxxxx@xxxx.xxx](#) , hace referencia al email del cliente.
- phone: valor numérico que hace referencia al número de teléfono del cliente.

Además de estos valores, se genera al momento de recibir el objeto los siguientes campos:

- token: código único que se genera al usuario para generar un link que pueda compartir para obtener referidos.
- referrals: valor numérico que se asigna en 0, hace referencia a la cantidad de usuarios que se han registrado con el link de referidos que se le genera al cliente.
- lastReferral: valor de tipo date con la fecha y hora del momento en que se genera.

En caso de que alguno de los valores no cumpla los requisitos se devuelve un error dependiendo del dato que no haya cumplido. Estas validaciones se realizan en `api/participants/models/participants.js`

```
validateUser: async ({ name, lastname, dni, email, phone, token, referrals, promotion }) => {
  const searchUser = strapi.services.participants.findOne
  if (name == null || name.length < 2) {
    const e = new Error('Nombre inválido')
    e.type = 'NOMBRE_INVALIDO'
    throw e
  } else if (lastname == null || lastname.length < 2) {
    const e = new Error('Apellido inválido')
    e.type = 'APELLIDO_INVALIDO'
    throw e
  } else if (dni == null || isNaN(dni) || dni < 100000 || dni > 99999999 || await searchUser({ dni: dni, promotion })) {
    const e = new Error('Dni inválido')
    e.type = 'DNI_INVALIDO'
    throw e
  } else if (email == null || (email.split('@')).length == 1 || await searchUser({ normalizedEmail: email.toUpperCase(), promotion })) {
    const e = new Error('Email inválido')
    e.type = 'EMAIL_INVALIDO'
    throw e
  } else if (phone == null || isNaN(phone) || await searchUser({ phone: phone, promotion })) {
    const e = new Error('Telefono inválido')
    e.type = 'TELEFONO_INVALIDO'
    throw e
  } else if (token == null || (await searchUser({ token: token, promotion })) != null) {
    const e = new Error('Token inválido')
    e.type = 'TOKEN_INVALIDO'
    throw e
  } else if (referrals == null || isNaN(referrals)) {
    const e = new Error('Referidos inválido')
    e.type = 'REFERIDOS_INVALIDO'
    throw e
  }
  return {
    name,
    lastname,
    dni,
    email,
    phone,
    token,
    referrals,
    promotion: promotion
  }
}
```

Esta ruta llama a ejecutar a la función **register** en `api\participants\controllers\participants.js` que se encarga de:

- Obtener del body de la petición un campo con los datos del usuario mencionados anteriormente, y un campo con el id de la promoción donde se está queriendo registrar el usuario.
- En caso de que no se reciba un ID de promoción devuelve un error indicando que el ID de la promoción es inválido.
- Si no encuentra cupones que tengan el ID de la promoción recibido, y además que en el atributo “used” tengan el valor “FALSE”, devuelve un error indicando que no hay cupones disponibles para la promoción.

- En ambos casos mencionados se interrumpe en el momento la función por lo cual no se realizaría ningún registro en el sistema con los datos enviados por el usuario.
- En caso de encontrar un cupón, continúa e intenta buscar en los parámetros recibidos de la petición si hay alguno llamado **referr**.
 - En caso de haberlo debería ser un código único correspondiente a un usuario ya registrado y se procede a buscar dicho usuario en el sistema.
 - De encontrarlo, aumenta en uno el valor que tenga ese usuario en el campo **referrals**.
 - En caso de no encontrar un usuario con ese token el flujo de registro continúa normalmente sin aumentar el campo **referrals** a ningún usuario.
- Una vez finalizado este paso, inserta los datos enviados por el usuario:
 - Se asigna un código único para el usuario en el campo **token** para poder generar el link que se le va a enviar junto con el cupón,
 - Se actualiza el campo “used” a “TRUE” del cupón obtenido anteriormente para evitar dárselo a otro usuario en el futuro.
 - Y envía una respuesta con código 201, que indica que se completó el flujo exitosamente, el usuario recibe el cupón para obtener su beneficio / descuento y el link de referidos que podrá compartir para aumentar la cantidad de puntos de su registro en el ranking.


```
async register(ctx) {
  try {
    const { user, promotionId } = await ctx.request.body;
    const coupon = await searchCouponByPromotionId(promotionId)
    const validUser = await customizeUser({ user, promotionId })
    const { referr } = ctx.query;

    if (referr) {
      updateReferral({ referr, promotionId })
    }
    await finishRegister({ validUser, coupon })
    ctx.send({
      status: 201,
      url_referrals: process.env.URL_LANDING + `?referr=${validUser.token}`,
      coupon: coupon.coupon
    });
  } catch (error) {
    ctx.send({
      status: 401,
      type: error,
      message: error.message
    });
  }
},
```

- GET /ranking

Para poder llegar a esta ruta se debe contar con el token de autenticación. Devuelve una lista de objetos que cumplen el siguiente modelo:

- Nombre: nombre del usuario.
 - Apellido: inicial del apellido del usuario.
 - Puntaje: la cantidad de usuarios que se han registrado usando el código de referido del usuario.
-
- En caso de recibir como parámetro un mail de un usuario registrado en el ranking devuelve la posición de dicho mail, de lo contrario devuelve 0 en este atributo de la respuesta.

Esta ruta llama a ejecutar a la función **getRanking** en `api\participants\controllers\participants.js` esta función se encarga de:

- Obtener los parámetros de la petición, uno llamado **email** y otro llamado **promotionId**.
- Ejecuta la función `searchPromotionById` que se encarga de buscar y devolver una promoción que con el ID recibido:
 - En caso de no encontrarla devuelve un error de ID inválido.
 - Si la obtiene exitosamente, extrae los participantes registrados en esa promoción y se los pasa como argumento a la función `sortUsers` que los ordena teniendo dos criterios:
 - el primero es el valor del campo `referrals` en orden descendente,
 - En caso de que dos o más registros tengan el mismo valor en este campo, el criterio de ordenamiento lo hace por el campo `lastReferral` de manera ascendente.
- Una vez ordenados verifica si el mail obtenido al principio tiene algún valor:
 - De tenerlo busca en la lista de usuario ya ordenada el índice donde se encuentra dicho registro,
 - En caso de no encontrarlo asigna un 0.
- Por último devuelve un código 200, la posición y la lista de usuarios ordenada y con una longitud máxima del campo `maxParticipants` de la promoción encontrada anteriormente.

```
async getRanking(ctx) {
  try {
    const { email, promotionId } = ctx.request.query

    const promotion = await searchPromotionById(promotionId)
    let users = promotion.participants

    users = users.sort(sortUsers)

    let position = 0
    if (email) {
      position = (users.findIndex((u) => u.email === email)) + 1
    }

    users = await validRanking({ users, promotion })

    ctx.send({
      status: 200,
      data: users,
      positionUserEmail: position
    })
  } catch (error) {
    ctx.send({
      status: 401,
      type: error,
      message: error.message
    });
  }
},
```

Promotions

Rutas

Ruta	Método	Función
/promotions	GET	find
/promotions/count	GET	count
/promotions/:id	GET	findOnePromotion
/promotions	POST	register
/promotions/:id	PUT	update
/promotions/:id	DELETE	delete

- GET /promotions/:id

Para poder acceder a esta ruta se debe contar con el token de autenticación y se debe recibir en el :id un id de promoción válido, de lo contrario se devolverá un error 404 por recurso no encontrado.

Esta ruta llama a ejecutar a la función **findOnePromotion** en la ruta `api/promotion/controller/promotion.js` y se encarga de:

- Buscar una promoción con el ID indicado

- Crea un valor booleano indicando si la fecha actual es mayor a la fecha en dateMax de la promoción
- Y asigna creando el campo expire
- Luego crea otro campo booleano y le asigna verdadero si encuentra cupones con el mismo ID de promoción que tengan en el campo “used” en valor “FALSE” llamado couponsAvaliables
- Finalmente se eliminan los campos que tienen la lista de cupones y participantes y se envía la promoción como respuesta.

```
async findOnePromotion(ctx) {
  try {
    const { id } = ctx.params;
    if (!id) {
      throw new Error(INVALID_PROMOTION_ID_MESSAGE)
    }
    const promotion = await strapi.services.promotion.findOne({ id })
    const today = new Date()
    const dateMax = new Date(promotion.dateMax)
    promotion.expired = dateMax < today
    promotion.coupons = promotion.coupons.filter(coupon => coupon.used === false)
    promotion.couponsAvaliables = promotion.coupons.length > 0
    delete promotion.coupons

    return promotion
  } catch (error) {
    if (error.message === INVALID_PROMOTION_ID_MESSAGE) {
      ctx.body = error.message
      ctx.status = 404
    } else {
      ctx.body = SERVER_UNEXPECTED_ERROR_MESSAGE
      ctx.status = 500
    }
  }
  ctx.send()
}
```

Coupons

Rutas

Ruta	Método	Función
/coupons	GET	find
/coupons/count	GET	count
/coupons/:id	GET	findOne
/coupons	POST	register
/coupons/:id	PUT	update
/coupons/:id	DELETE	delete

Para realizar a carga de los cupones de manera más rápida se recomienda usar un ciclo como el siguiente, reemplazando en la constante auth por un JWT registrado, asignando en coupons un array de strings con los valores que se les quiere cargar a los cupones, cambiar en data.promotion por el ID correspondiente a la promoción donde se quieren cargar los cupones:

```
const auth = {
  headers: {
    Authorization:
      `Bearer ` // AQUI CONCATENAR EL TOKEN DE AUTENTICACIÓN VALIDO
  },
}

const data = {
  promotion: '234567890' // ID DE LA PROMOCIÓN A LA QUE SE QUIERAN CARGAR LOS CUPONES
}
const coupons = [] // Array con los String que se le quiere asignar a cada cupon

let i = 0 // CANTIDAD DE CUPONES A CARGAR
while (i <= coupons.length) {
  data.coupon = coupons[i]

  axios.post('http://localhost:1337/coupons', data, auth) // REEMPLAZAR LOCALHOST:1337 POR LA URL/IP CORRESPONDIENTE
  i++
}
```

Base de Datos

El archivo que se encarga de la base de datos se encuentra en /config/database.js.

El mismo está diseñado para utilizar una conexión con **POSTGRESQL**, sin embargo Strapi ofrece varias opciones de conexión a la base de datos. Se deberá tener creada una base de datos correspondiente al nombre indicado en la variable de entorno DATABASE_NAME , no es necesario crear las tablas ya que la primera vez que se ejecutó el proyecto se crearán tanto las tablas como sus relaciones.

En caso de querer cambiar o migrar a otra tecnología se puede seguir la documentación oficial:

<https://strapi.io/documentation/developer-docs/latest/setup-deployment-guides/configuration.html#database>

```

1  module.exports = ({ env }) => ({
2    defaultConnection: 'default',
3    connections: {
4      default: {
5        connector: 'bookshelf',
6        settings: {
7          client: 'postgres',
8          host: env('DATABASE_HOST'),
9          port: env.int('DATABASE_PORT'),
10         database: env('DATABASE_NAME'),
11         username: env('DATABASE_USERNAME'),
12         password: env('DATABASE_PASSWORD'),
13         ssl: false,
14       },
15       options: {},
16     },
17   },
18   });

```

Códigos HTTP y Errores

La siguiente tabla muestra los posibles códigos y errores HTTP:

Código	Descripción	Error	Solución
200	La petición se realizó de manera satisfactoria	No	N/A
201	La petición se realizó de manera satisfactoria, se agregó o modificó	No	N/A

	uno o más registros.		
401	No se encontró la ruta o recurso especificado	Si	Verificar la ruta o recurso solicitado.
404	Error de autenticación contra el servidor	Si	Verificar el token de autenticación.
500	Error interno del servidor o de la base de datos.	Si	Verificar el mensaje de error devuelto con la petición

Introducción a React

Como ya hemos mencionado anteriormente, React es una librería JavaScript focalizada en el desarrollo de interfaces de usuario. Facilita la creación de componentes interactivos y reutilizables. Ofrece grandes beneficios en performance, modularidad y promueve un flujo muy claro de datos y eventos, facilitando la planeación y el desarrollo web.

Para tener una performance muy alta, React implementa algo llamado Virtual DOM. En vez de renderizar todo el DOM en cada cambio, que es lo que normalmente se hace, React hace los cambios en una copia en memoria y después usa un algoritmo para comparar las propiedades de la copia en memoria con las de la versión del DOM y así aplicar cambios exclusivamente en las partes que varían.

Variables de entorno

Antes de poder ejecutar el proyecto se deben tener definidas las siguientes variables de entorno:

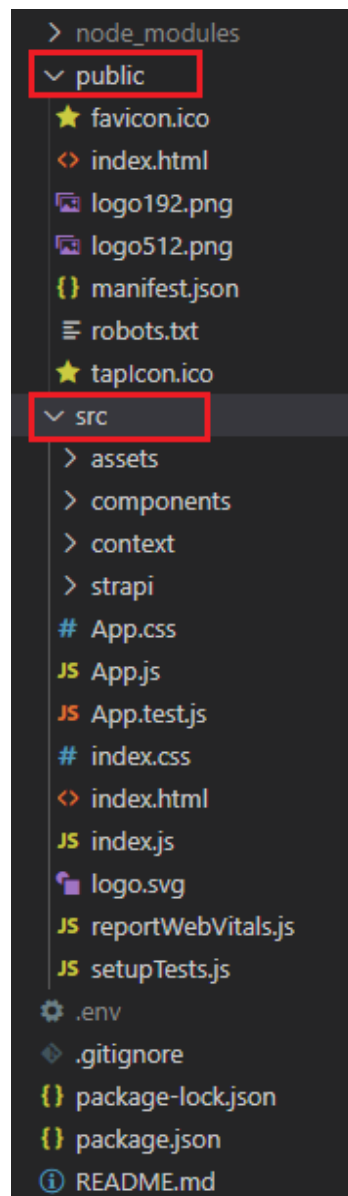
REACT_APP_AUTHORIZATION_STRAPI: string que contiene el JSON Web Token para poder tener la autorización de uso de Strapi

REACT_APP_URL_STRAPI: string que contiene la URL de Strapi.

REACT_APP_PROMOTION_ID: el mismo contiene el ID de la promoción que se quiera utilizar al momento de darle vigencia a una campaña.

Distribución de carpetas

A continuación se mostrarán las carpetas y archivos utilizados a lo largo de todo el proyecto:

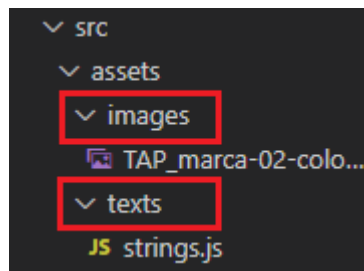


Podemos apreciar que tenemos:

Dos carpetas principales:

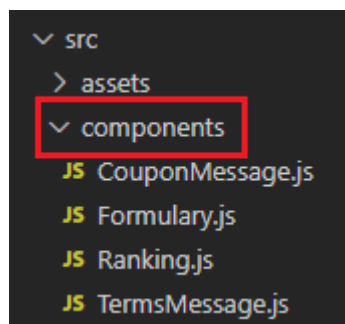
- Carpeta **public**: donde almacenamos todos los archivos de acceso público.
- Carpeta **src**: donde almacenamos todas las carpetas y archivos privados que componen a la landing web. Veremos que dentro de él tenemos los siguientes archivos importantes y relevantes que hacen al negocio:

- Carpeta assets:



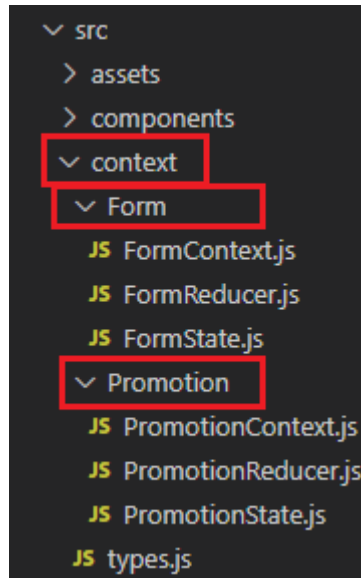
Tendremos almacenados la imagen y los textos utilizados en el proyecto.

- Carpeta components:



Tendremos almacenados los archivos '.js'/componentes que hacen al proyecto.

- Carpeta context:



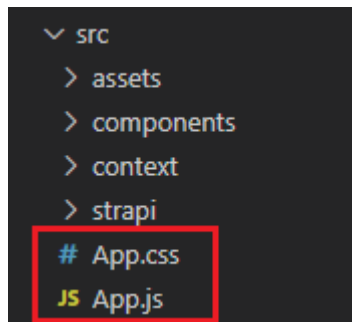
Tendremos almacenados los archivos '.js' y las carpetas que contienen los contextos utilizados en todo el proyecto para el uso de datos globales por cada componente presente en el mismo.

- Carpeta strapi:



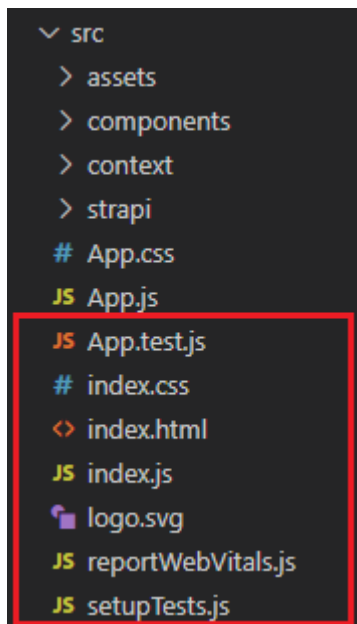
Tendremos almacenados los archivos '.js' que nos permiten obtener los datos desde strapi (data.js) y el uso de variables de entorno para poder acceder a los mismos (config.js)

- Archivo App.js y App.css:

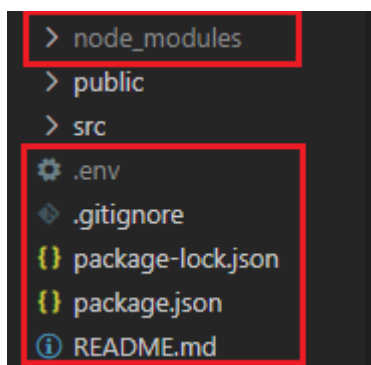


Tendremos la base de los estilos (App.css) y el esqueleto con todos los componentes anteriormente tratados en la carpeta "components" (App.js), de la Landing Web.

- Archivos extras: generados por 'create-react-app':



- Archivos y carpetas secundarias (no menos importantes):



En este apartado encontraremos el archivo para las variables de entorno (.env), el archivo para la exclusión de otros archivos/carpetas a la hora de subir el código a GitHub (.gitignore), los respectivos metadatos que contienen los archivos package.json y la carpeta de dependencias node_modules.

Componentes

A continuación se mostrarán los componentes utilizados a lo largo de todo el proyecto:

Formulary

El componente Formulary es utilizado para generar un formulario que le permite al Usuario registrarse en la campaña vigente. El mismo cuenta con dos pasos en los cuales el Usuario debe ingresar sus datos personales y a su vez aceptar un acuerdo de términos y condiciones.

A continuación mostraremos y explicaremos el código empleado:

- **Importaciones:**

```
1 import React, { useContext } from 'react'
2 import { texts } from '../assets/texts/strings.js'
3 import { Card, CardContent, CircularProgress, Button, Box, Stepper, Step, StepLabel, Grid, Typography } from '@material-ui/core'
4 import { Field, Form, Formik, ErrorMessage } from 'formik'
5 import { TextField } from 'formik-material-ui'
6 import FormContext from '../context/Form/FormContext.js'
7 import * as Yup from 'yup';
8 import swal from 'sweetalert';
9 import { useLocation } from "react-router-dom";
10 import { Scrollbar } from 'react-scrollbars-custom'
11 import { getCoupon } from '../strapi/data.js'
```

Podemos apreciar que importamos:

- El hook de **'useContext'**: para poder acceder y utilizar variables globales.

- Objeto **texts**: para acceder a los textos necesarios para cada componente.
- Componentes de **MaterialUI**: para poder aplicar estilos.
- Componentes de la librería '**Formik**': para poder manejar el formulario.
- Componente de contexto '**FormContext**': para también poder acceder a las variables globales del formulario.
- Librería **Yup**: para poder realizar las validaciones del formulario.
- Librería **sweetalert**: para poder realizar las alertas de error correspondientes.
- Librería **react-scrollbar-custom**: para poder agregar una barra deslizable al componente de formulario.
- Función **getCoupon()**: para poder obtener el cupón de descuento que se le ofrece al Usuario.

- **Componente Formulary:**

```
14 export default function Formulary() {  
15   const { dataForm, setRegisteredUser, setStep, setFormCompleted, setCoupon, setOpenCouponModal, setOpenTermsModal } = useContext(FormContext);  
16   const handleOpenCoupon = () => setOpenCouponModal(true)  
17   const handleOpenTerms = () => setOpenTermsModal(true)  
18  
19   function useQuery() {  
20     return new URLSearchParams(useLocation().search)  
21   }  
22  
23   const query = useQuery();
```

Podemos apreciar:

El uso de las variables y funciones globales del Formulario con FormContext. Las mismas serán usadas durante todo el componente:

- **dataForm:** objeto para inicializar los valores que recibiremos del formulario con Formik.
 - **setRegisteredUser:** función para confirmar si el usuario fue registrado o no. Si el valor de la variable es true, es usado en Ranking para actualizar la lista de participantes.
 - **setStep:** función usada para actualizar el paso del formulario a cero (al primero) luego de que el usuario se registró.
 - **setFormCompleted:** función usada para confirmar si el formulario fue completado o no, es usada en el gráfico de los pasos a seguir. Si está en true, el icono de los pasos completados aparece; si no lo está, siguen figurando sus respectivos números.
 - **setCoupon:** función usada para setear un objeto de cupón de descuento luego de que un usuario se haya registrado.
 - **setOpenCouponModal:** junto con la función handleOpenCoupon(), son las encargadas de setear en true la variable booleana global openCouponModal para mostrar un mensaje en una ventana modal pop-up luego de que se haya registrado una persona.
 - **setOpenTermsModal:** junto con la función handleOpenTerms(), son las encargadas de setear en true la variable booleana global openTermsModal para mostrar un mensaje en una ventana modal pop-up sobre términos y condiciones de la campaña vigente.
 - **Función useQuery():** utilizada para obtener los parámetros de la query de la URL.
-
- Función request():

```
25 const request = async (values) => {
26   const user = {
27     name: values.name,
28     lastname: values.lastName,
29     dni: values.dni,
30     email: values.email,
31     phone: values.phone,
32   }
33
34   const req = { user }
35
36   try {
37     const data = await getCoupon({ referr: query.get("referr"), req })
38
39     if (data != null) {
40       setCoupon({ ...data })
41       handleOpenCoupon()
42       setFormCompleted(true)
43       setRegisteredUser(true)
44     }
45   } catch (error) {
46     swal({
47       title: "¡Error!",
48       text: `${error.message}`,
49       icon: 'error',
50       button: {
51         text: "Aceptar",
52       },
53       timer: 10000
54     });
55   }
56 }
```

La función **request()** es utilizada luego de que el usuario termina de registrarse en el formulario. Almacena todos los datos del usuario en un objeto y junto a él hace una petición para obtener el cupón de descuento correspondiente:

- **Si el mismo es devuelto:**
 - Setea el cupón en una variable global “coupon” para que pueda ser accedido por otro componente (CouponMessage)
 - Setea en ‘true’ la variable booleana global, del modal/pop-up del cupón obtenido, con `handleOpenCoupon()`, para que aparezca y muestre el mensaje consagratorio

- Setea en 'true' las variables globales `registeredUser` y `formCompleted` con `setRegisteredUser()` y `setFormCompleted()`, explicadas anteriormente.
- Si el mismo no es devuelto: captura el error y genera una ventana/pop-up indicando al usuario que ha ocurrido un error.
- Finalmente, el componente `Formulary` **retorna** lo siguiente:

```

57     return (
58       <Grid item style={{ ...styles.gridContainer, height: '80vh', width: '100%' }}>
59         <Card id="form-card-container">
60           <Scrollbar style={{ width: '100%', height: '100%', zIndex: 300 }}>
61             <CardContent>
62               <FormikStepper>
63                 <InitialValues={{ ...dataForm }}>
64                   <onSubmit={async (values, helpers) => {
65                     await request(values);
66                     helpers.resetForm();
67                     setStep(0);
68                     setFormCompleted(false);
69                     setRegisteredUser(false);
70                   }}>
71                     <div title="Datos personales">
72                       <Box paddingBottom={1.5}>
73                         <Field fullWidth name="nombre" component={TextField} label={texts.FORM_LABEL_NAME} variant="outlined" InputLabelProps={{ id: 'form-label-name', style:
74                           styles.textField }} />
75                       </Box>
76                       <Box paddingBottom={1.5}>
77                         <Field fullWidth name="apellido" component={TextField} label={texts.FORM_LABEL_LASTNAME} variant="outlined" InputLabelProps={{ id: 'form-label-last-name',
78                           style: styles.textField }} />
79                       </Box>
80                       <Box paddingBottom={1.5}>
81                         <Field fullWidth type="number" name="dni" component={TextField} label={texts.FORM_LABEL_DNI} variant="outlined" InputLabelProps={{ id: 'form-label-dni',
82                           style: styles.textField }} />
83                       </Box>
84                     </div>
85                     <div title="Contacto">
86                       <Box paddingBottom={2}>
87                         <Field type="email" fullWidth name="email" component={TextField} label={texts.FORM_LABEL_EMAIL} variant="outlined" InputLabelProps={{ id:
88                           'form-label-email', style: styles.textField }} />
89                       </Box>
90                       <Box paddingBottom={2}>
91                         <Field fullWidth type="number" name="telefono" component={TextField} label={texts.FORM_LABEL_PHONE} variant="outlined" InputLabelProps={{ id:
92                           'form-label-phone', style: styles.textField }} />
93                       </Box>
94                     </div>
95                   </FormikStepper>
96                 </CardContent>
97             </Scrollbar>
98           </Card>
99         </Grid>
100       </div>
101     );

```

```

102       <Box paddingBottom={2} style={{textAlign: 'center'}}>
103         <label id="form-label-terms" style={{ ...styles.textField }}>
104           <Field
105             type="checkbox"
106             name="acceptTerms"
107           />
108           <{texts.FORM_LABEL_TERMS} <Button id="form-button-3" variant="text" onClick={handleOpenTerms} style={{ color: '#002350', backgroundColor: '#F3F3F3' }}>
109             términos y condiciones.</Button> </>
110         </label>
111         <ErrorMessage name="acceptTerms" component="div" style={{ color: 'red', fontSize: '13px', paddingLeft: '15px' }} />
112       </Box>
113     </FormikStepper>
114   </CardContent>
115 </Scrollbar>
116 </Card>
117 </Grid>
118 )
119

```

Como podemos apreciar, dentro del **'return'** del componente `Formulary`, nos podemos encontrar con el formulario que el Usuario utilizará para poder registrarse

en la campaña vigente. Donde deberá ingresar su nombre, apellido, dni, email, teléfono y una conformidad sobre términos y condiciones.

Formulary vendrá acompañado de componentes tales como MaterialUI, ScrollBar Custom y Formik que le darán estilo y funcionalidad:

- **Grid:** Cuadrícula que permite flexibilidad en una amplia variedad de diseños.
 - **Card:** Superficie que muestra contenido sobre un solo tema.
 - **CardContent:** Permite mostrar el contenido deseado.
 - **Scrollbar:** Como el nombre lo dice, una barra de scroll que permite deslizarse cuando las pantallas son reducidas y el contenido debe ajustarse.
 - **Box:** Componente contenedor.
 - **TextField:** para permitir a los usuarios ingresar y editar texto.
 - **ErrorMessage:** para mostrar un mensaje de error en caso de que el Usuario no haya dado conformidad sobre términos y condiciones y quiera continuar.
 - **Field:** Conecta automáticamente las entradas del Usuario a Formik. Utiliza el atributo 'name' para que coincida con el estado inicial de Formik y así poder asignar cada valor a la propiedad indicada.
-
- **Componente FormikStepper:**

```

112 export function FormikStepper({ children, ...props }) {
113   const { step, setStep, formCompleted } = useContext(FormContext);
114   const childrenArray = React.Children.toArray(children)
115   const currentChild = childrenArray[step]
116
117   const isLastStep = () => {
118     return step === childrenArray.length - 1;
119   }
120
121   return (
122     <Formik
123       {...props}
124       onSubmit={async (values, helpers) => {
125         if (isLastStep()) {
126           await props.onSubmit(values, helpers)
127         } else {
128           setStep(step + 1)
129           helpers.setTouched({});
130         }
131       }}
132       validationSchema={isLastStep()
133         ? Yup.object({
134             name: Yup.string().max(20, `${texts.ONLY_ALPHABET_TEXT_VAL_NAME}`).min(2, `${texts.ONLY_ALPHABET_TEXT_VAL_NAME}`).matches(/^[a-z_]+([a-z_]+)*$/gim, `${texts.ONLY_ALPHABET_TEXT_VAL_NAME}`).required(`${texts.REQUIRED_TEXT_VAL`),
135             lastName: Yup.string().max(20, `${texts.ONLY_ALPHABET_TEXT_VAL_LASTNAME}`).min(2, `${texts.ONLY_ALPHABET_TEXT_VAL_LASTNAME}`).matches(/^[a-z]([a-z](0,20){[a-z]+)*$/gim,
136             dni: Yup.number().integer("No puede ingresar valores con puntos o comas.").positive("No puede ingresar valores negativos.").lessThan(100000000, `${texts.MIN_NUMBER_DNI_VAL`
137             }).moreThan(10000000, `${texts.MAX_NUMBER_DNI_VAL`}).required(`${texts.REQUIRED_TEXT_VAL`})
138           })
139         : Yup.object({
140             email: Yup.string().email(`${texts.EMAIL_VAL`}).required(`${texts.REQUIRED_TEXT_VAL`}),
141             phone: Yup.number().integer("No puede ingresar valores con puntos o comas.").positive("No puede ingresar valores negativos.").lessThan(9999999999, `${texts.PHONE_VAL`}).
142             moreThan(1100000000, `${texts.PHONE_VAL`}).required(`${texts.REQUIRED_TEXT_VAL`}),
143             acceptTerms: Yup.boolean().oneOf([true], "No es posible dejar el campo sin completar.").required(`${texts.REQUIRED_TEXT_VAL`})
144           })
145       )
146     >

```

```

145   ({ isSubmitting }) => {
146     <Form autoComplete="off">
147       <Grid container direction="column" justifyContent="center" alignContent="center">
148         <Grid item container alignContent="center" justifyContent="center">
149           <Stepper activeStep={step} alternativeLabel>
150             {childrenArray.map((child, index) => {
151               <Step key={child.props.title} completed={step > index || formCompleted}>
152                 <StepLabel>
153                   <Typography id="label-step">{child.props.title}</Typography>
154                 </StepLabel>
155               </Step>
156             )}
157           </Stepper>
158         </Grid>
159         <Grid item>
160           {currentChild}
161         </Grid>
162         <Grid item container spacing={2} alignContent="center" justifyContent="center">
163           <Grid item>
164             {step > 0
165               ?
166               <Button
167                 id="form-button-1"
168                 disabled={isSubmitting}
169                 variant="contained"
170                 style={{ backgroundColor: isSubmitting ? '#CDCDCD' : '#4E53DB', color: isSubmitting ? '#757575' : '#FFFFFF' }}
171                 onClick={() => setStep(step - 1)}
172                 {texts.BACK_BUTTON}
173               </Button>
174             : null}
175           </Grid>
176           <Grid item>
177             <Button
178               id="form-button-2"
179               disabled={isSubmitting} variant="contained"
180               startIcon={isSubmitting ? <CircularProgress size="1rem" /> : null}
181               style={{ backgroundColor: isSubmitting ? '#CDCDCD' : '#4E53DB', color: isSubmitting ? '#757575' : '#FFFFFF' }} type="submit">
182                 {texts.NEXT_BUTTON}
183               </Button>
184             </Grid>
185           </Grid>
186         </Grid>
187       </Form>
188     </Formik>
189   )
190 }
191 </Formik>

```

El componente FormikStepper contiene:

- **FormContext:** para hacer uso de las funciones y estados globales que contiene. En este caso, hacemos uso del paso actual del formulario (debido a que el mismo cuenta con dos pasos) y si fue completado o no.

- **currentChild:** constante que contiene el 'div' del paso actual del formulario, es decir, como vimos que anteriormente el formulario se divide en dos 'div's', esta constante contiene el 'div' correspondiente al paso actual.
- **isLastStep:** función que, como su nombre lo indica, es usada para corroborar si el paso actual es el último.

El componente FormikStepper retorna:

El componente Formik, que es el encargado del completo manejo del formulario, es decir: el manejo de envío del mismo, el manejo de los valores/estados y de las validaciones y mensajes de error (junto a otro Componente llamado "Yup"). A su vez, como vemos, en sus propiedades contiene:

- **initialValues:** propiedad que obtiene del pasaje de propiedades al utilizar el componente tratado (FormikStepper). El mismo contiene los valores iniciales de las propiedades a utilizar en el formulario (nombre, apellido, dni, email, teléfono y conformidad de los términos), gracias a este objeto, Formik puede asignar los registros que el Usuario haya ingresado.
- **onSubmit():** función que permite el manejo del envío del formulario, la misma, al igual que initialValues obtiene del pasaje de propiedades el siguiente proceso de invocación a funciones:
- **request():** función que almacena todos los datos del usuario en un objeto para que junto a él se haga una petición para obtener el cupón de descuento correspondiente, la misma se explicó en más detalle anteriormente.
- **helpers.resetForm():** función que sirve para resetear los campos del formulario así el usuario no sigue viendo los registros que fue ingresando.

- **setStep(0):** función usada para que cuando el usuario haya terminado su proceso de registro, el formulario vuelva al primer paso.
- **setFormCompleted():** función usada para confirmar si el formulario fue completado o no.
- **setRegisteredUser():** función usada para confirmar si el usuario se registró correctamente.

Además, valida que si el paso actual es el último de los pasos, el envío del formulario debe concretarse y si no es el último paso, que al terminar de completar los registros, se pase al siguiente paso seteando el paso actual global (step) con "+1". Finalmente, permite con 'helpers.setTouched({})', que el siguiente paso del formulario tenga la posibilidad de validar cada uno de sus campos

- **validationSchema:** junto a Yup y expresiones regulares, se hacen las siguientes validaciones a los campos del formulario:
 - **name:** donde se valida que el mismo sea solo texto, con espacios permitidos solo entre palabras, con un largo mínimo de 2 caracteres y uno máximo de 20 caracteres. Este campo es requerido.
 - **lastName:** donde se valida que el mismo sea solo texto, con espacios permitidos solo entre palabras, con un largo mínimo de 2 caracteres y uno máximo de 20 caracteres y con la posibilidad de colocar un apóstrofe en la 2da posición. Este campo es requerido.
 - **dni:** donde se valida que el mismo sea solo números enteros positivos, con un largo mínimo de 10000000 números y uno máximo de 100000000 números. Este campo es requerido.

- **email:** donde se valida que el mismo respete tener al menos el siguiente formato 'texto@texto.texto'. Este campo es requerido.
- **phone:** donde se valida que el mismo sea solo números enteros positivos, con un largo mínimo de 1100000000 números y uno máximo de 9999999999 números. Este campo es requerido.
- **acceptTerms:** donde se valida que el mismo sea 'true'. Este campo es requerido.

Para finalizar, el componente FormikStepper, además de retornar el componente Formik, como es debido, **retorna el formulario que el Usuario utilizara para su interacción y registro**. El mismo tendrá:

- Un indicador del paso en el que el Usuario se encuentra.
- Los campos que el Usuario debe completar de acuerdo con el paso vigente.
- Un botón: para continuar, si el Usuario se encuentra en un primer paso.
- Dos botones: uno para volver (para regresar al paso anterior) y el otro para continuar (para el envío del formulario) si el Usuario se encuentra en el último paso.

Finalmente, el componente Formulary, tendrá los siguientes estilos que serán usados a lo largo de todo el componente:

```
196 const styles = {
197   gridContainer: {
198     justifyItems: 'center',
199     alignItems: 'center',
200     justifyContent: 'center',
201     alignContent: 'center'
202   },
203   borderCoupon: {
204     borderRadius: '50px',
205     borderColor: '#14D2B9',
206     borderWidth: '3px',
207     borderStyle: 'solid',
208     margin: '10px auto',
209     padding: '10px'
210   },
211   buttonCoupon: {
212     backgroundColor: '#14D2B9',
213     color: 'FFFFFF'
214   },
215   copiedText: {
216     color: 'FFFFFF',
217     fontSize: '13px',
218     margin: '5px auto'
219   },
220   textField: {
221     fontSize: '14px'
222   }
223 }
```

Ranking

El componente Ranking es utilizado para generar un Ranking, valga la redundancia, donde se podrá ver la lista de participantes, sus posiciones y sus premios obtenidos al momento. Además se podrá consultar la posición y premio obtenido por separado.

A continuación mostraremos y explicaremos el código empleado:

- **Importaciones:**

```
1 import React, { useState, useEffect, useContext } from 'react'
2 import { texts } from '../assets/texts/strings.js'
3 import { Button, Modal, Box, Card, CardContent, Grid, Typography, List, ListItem, CircularProgress } from '@material-ui/core'
4 import ArrowBackIcon from '@mui/icons-material/ArrowBack';
5 import IconButton from '@mui/material/IconButton';
6 import { Scrollbar } from 'react-scrollbars-custom'
7 import FormContext from '../context/Form/FormContext.js'
8 import PromotionContext from '../context/Promotion/PromotionContext.js'
9 import { getAllParticipants } from '../strapi/data.js'
10 import { Field, Form, Formik } from 'formik'
11 import { TextField } from 'formik-material-ui'
12 import * as Yup from 'yup';
13 import swal from 'sweetalert';
14 import { getPromotionId } from '../strapi/config.js';
```

- Podemos apreciar que importamos:
 - El hook de **'useState'**: para poder hacer uso de un estado en un componente de función.
 - El hook de **'useEffect'**: para poder realizar operaciones secundarias que impliquen, por ejemplo, una petición de datos.
 - El hook de **'useContext'**: para poder acceder y utilizar variables globales.
 - Objeto **texts**: para acceder a los textos necesarios para cada componente.
 - Componentes de **MaterialUI**: para poder aplicar estilos.
 - Icono **'ArrowBackIcon'** para poder volver atrás en cada modal.
 - Librería **react-scrollbars-custom**: para poder agregar una barra deslizable al componente.
 - Componente de contexto **'FormContext'**: para también poder acceder a las variables globales del formulario.
 - Componente de contexto **'PromotionContext'**: para también poder acceder a las variables globales de la promoción vigente.
 - Función **'getAllParticipants'**: para poder realizar una petición que nos permite obtener la lista de participantes de la campaña.

- Componentes de la librería **'Formik'**: para poder manejar el formulario.
 - Librería **Yup**: para poder realizar las validaciones del formulario.
 - Librería **sweetalert**: para poder realizar las alertas de error correspondientes.
 - Función **getPromotionId()**: para poder obtener el id de promoción de la promoción vigente.
- **Componente Ranking:**

```
17  const Ranking = () => {
18    const { promotion } = useContext(PromotionContext);
19    const { registeredUser } = useContext(FormContext);
20    const [participants, setParticipants] = useState(null);
21    const [userPositionRanking, setUserPositionRanking] = useState(0);
22    const [isCompleted, setIsCompleted] = useState(false);
23    const [open, setOpen] = useState(false);
24    const handleOpen = () => setOpen(true);
25    const handleClose = () => {
26      setOpen(false)
27      setIsCompleted(false)
28    };
29
30
31    useEffect(() => {
32
33      async function fetchMyAPI() {
34        await getAllParticipants({ promotionId: getPromotionId() })
35          .then(({ data }) => setParticipants(data))
36          .catch((e) => {
37            swal({
38              title: "¡Error!",
39              text: `${texts.ERROR_NO_REGISTERED_USERS}`,
40              icon: 'error',
41              button: {
42                text: "Aceptar",
43              },
44              timer: 10000
45            });
46          })
47      }
48
49      fetchMyAPI()
50
51    }, [registeredUser]);
```

Podemos apreciar:

- El uso del objeto global **promotion**, que va a ser utilizado durante todo el componente para el acceso a sus propiedades que describen toda la promoción vigente, es decir, los precios que maneja, las descripciones de la misma, las condiciones que maneja, etc.
 - El uso de la variable booleana **registeredUser** para, en el caso de que esté en 'true', actualizar la lista de participantes.
 - Un hook de estado '**participants**' para almacenar y acceder a la lista de usuarios obtenidas en la petición realizada.
 - Un hook de estado '**userPositionRanking**' para almacenar el estado de la posición en Ranking que fue consultada por el Usuario.
 - Un hook de estado '**isCompleted**' para saber cuando la petición de la posición del Usuario fue realizada y cuando acabó.
 - Un hook de estado '**open**' que junto a la función **setOpen** (manejada por **handleOpen** y **handleClose**) nos permite indicar cuándo abrir una ventana modal y cuando cerrarla.
 - El uso del hook **useEffect()** para hacer peticiones de la lista de participantes cada vez que haya un nuevo Usuario registrado y así poder actualizar la misma en tiempo real. Hacemos una captura de los datos obtenidos en el caso de que todo salga bien y una captura del error en caso de que se presente algún inconveniente.
-
- Función **renderParticipantDetail**:

```
53 function renderParticipantDetail() {
54   return (
55     <Scrollbar style={{ height: '300px' }}>
56       <List>
57         {
58           participants.length > 0
59           ?
60             participants.map(function (participante, pos) {
61               return (
62                 <ListItem className='ranking-list-item' key={pos}>
63                   <Grid item xs=(2) sm=(2) md=(2)>
64                     <div className='pos-item-circle'>
65                       <Typography id='pos-part-ranking' style={{ color: '#405A7C' }}>{pos + 1}</Typography>
66                     </div>
67                   </Grid>
68                   <Grid item xs=(7) sm=(8) md=(8) style={{ textAlign: 'center', flexDirection: 'row' }}>
69                     <Typography
70                       nowrap
71                       id='name-part-ranking' style={{ color: 'FFFFFF' }}>{`${participante.name}`}</Typography>
72                     <Typography
73                       id='name-part-ranking' style={{ display: 'inline', color: 'FFFFFF' }}>{`${participante.lastname[0]}.`}</Typography>
74                     </Grid>
75                     <Grid item xs=(3) sm=(2) md=(2) style={{ textAlign: 'center' }}>
76                       <Typography id='points-part-ranking' style={{ color: '#6EF1C7' }}>{`${participante.referrals} puntos`}</Typography>
77                     </Grid>
78                   </ListItem>
79                 )
80             )
81           : null
82         }
83       </List>
84     </Scrollbar>
85   )
86 }
87
88
89
90 }
```

La función `renderParticipantDetail()` retorna la lista de participantes vigentes en el caso de que los haya. Cada participante se renderiza con su posición, nombre y apellido (solo primera letra del apellido) y premio a obtener por posición.

Se utilizan componentes de MaterialUI para darle un estilo y funcionalidad a la lista:

- **Scrollbar:** Como su nombre lo indica una barra de scroll que permite deslizarse cuando las pantallas son reducidas y el contenido debe ajustarse.
- **List:** componente contenedor que nos permite generar una lista.
- **ListItem:** componente para cada ítem de la lista vigente.
- **Grid:** Cuadrícula que permite flexibilidad en una amplia variedad de diseños.
- **Typography:** componente que representa contenido textual.

- Función **ChildModal**:

```

92  const ChildModal = () => {
93    const [openModal, setOpenModal] = useState(false);
94    const handleOpen = () => setOpenModal(true);
95    const handleClose = () => setOpenModal(false);
96
97    return (
98      <>
99        <Button id='ranking-button-2' variant='contained' onClick={handleOpen}>{texts.PRIZES_BUTTON}</Button>
100        <Modal
101          open={openModal}
102          onClose={handleClose}
103          style={{ display: 'flex', alignItems: 'center', justifyContent: 'center' }}
104        >
105          <Card id='ranking-card-2' className='unselectable'>
106            <Scrollbar>
107              <CardContent>
108                <Grid container spacing={3} style={styles.gridContainer}>
109
110                  <Grid item container direction='row' style={{ ...styles.gridContainer, marginTop: '30px', justifyContent: 'center', height: '5vh' }}>
111                    <Grid item container style={{ justifyContent: 'space-between' }}>
112                      <Grid item>
113                        <IconButton aria-label="Volver" sx={{ color: '#FFFFFF' }} onClick={handleClose}>
114                          <ArrowBackIcon />
115                        </IconButton>
116                      </Grid>
117                    </Grid>
118                  </Grid>
119
120                  <Grid item style={{ ...styles.gridContainer, textAlign: 'center' }}>
121                    <Typography id='prizes-title' style={styles.textTitle}>{texts.TITLE_OF_PRIZES}</Typography>
122                  </Grid>
123
124                  <Grid item container style={styles.gridContainer}>
125                    <Card style={{ backgroundColor: '#002350', borderColor: '#140289', borderStyle: 'solid', borderWidth: '5px' }}>
126                      <CardContent>
127                        <Grid item style={{ textAlign: 'center', padding: '10px' }}>
128                          <Typography id='part-prize-price-1' style={styles.textDescription}>{promotion.descriptionFirstPrize} ${promotion.prizeMaxPrice}</Typography>
129                          <Typography id='part-prize-price-2' style={styles.textDescription}>{promotion.descriptionSecondPrize} ${promotion.prizeMinPrice}</Typography>
130                        </Grid>
131                      </CardContent>
132                    </Card>
133                  </Grid>
134                </Grid>
135              </CardContent>
136            </Scrollbar>
137          </Card>
138        </Modal>
139      </>
140    );
141  };
142
143  <Grid item container direction='column' style={{ justifyContent: 'center', alignContent: 'center' }}>
144    {participants != null && participants.length > 0
145      ?
146        renderParticipantDetail()
147      :
148        <Grid item style={{ textAlign: 'center', padding: '10px', marginTop: '30px' }}>
149          {promotion.expired == false && promotion.couponsAvailable
150            ?
151              <Typography id='no-participants' style={{ ...styles.textDescription, fontSize: '1.5rem' }}>{texts.NO_PARTICIPANTS}</Typography>
152            :
153              <Typography id='no-participants' style={{ ...styles.textDescription, fontSize: '1.5rem' }}>{texts.RANKING_WITHOUT_COUPONS}</Typography>
154            }
155          </Grid>
156        </Grid>
157    }
158  </Grid>
159 </Grid>
160 </CardContent>
161 </Scrollbar>
162 </Card>
163 </Modal>
164 </>
165 );
166 }

```

La función **ChildModal**, retorna un componente de ventana **Modal** tipo **pop-up** que nos indica los premios vigentes y la lista de participantes del **Ranking**. En el caso de que no haya participantes se muestra un mensaje que indica que aún no los hay.

Se utilizan componentes de **MaterialUI** para darle un estilo y funcionalidad a la ventana.

- **Button:** Como su nombre lo indica, un botón que permite al Usuario acceder a este mismo componente Modal.
- **Modal:** componente que proporciona una base sólida para ventanas pop-up. El mismo cuenta con la propiedad “open” que utiliza un estado booleano para saber cuándo abrirse y “onClose” que es un evento que junto a handleClose() permiten al Usuario salir del Modal tocando todo lo externo a él, es decir, su fondo.
- **Scrollbar:** Como su nombre lo indica, una barra de scroll que permite deslizarse cuando las pantallas son reducidas y el contenido debe ajustarse.
- **Card:** Superficie que muestra contenido sobre un solo tema.
- **CardContent:** Permite mostrar el contenido deseado.
- **Grid:** Cuadrícula que permite flexibilidad en una amplia variedad de diseños.
- **IconButton:** componente que permite (junto a ArrowBackIcon) representar un ícono para ‘volver’.
- **Typography:** componente que representa contenido textual.

Función **request()**:


```
160     const request = async (email) => {
161
162         try {
163             const { positionUserEmail } = await getAllParticipants({ promotionId: getPromotionId(), email })
164             setIsCompleted(true)
165             return positionUserEmail
166         } catch (error) {
167             swal({
168                 title: "¡Error!",
169                 text: `${texts.ERROR_NO_REGISTERED_USERS}`,
170                 icon: 'error',
171                 button: {
172                     text: "Aceptar",
173                 },
174                 timer: 10000
175             });
176         }
177     }
```

La función request() es usada luego de que el usuario termina de consultar en el formulario del Ranking con su email su posición. Al recibir el email por argumento, realiza las peticiones correspondientes y dependiendo del valor recibido...:

- Si el mismo **es devuelto**: almacena los datos del participante un objeto y junto a él setea en 'true' el hook isCompleted para poder saber que el Usuario acaba de hacer la consulta.
- Si el mismo **no es devuelto**: captura el error y genera una ventana/pop-up indicando al usuario que ha ocurrido un error.

Finalmente el componente Ranking **retorna**:

```

180     return (
181       <>
182         <div className='ranking-button-container'>
183           <Button id='ranking-button' variant='contained' onClick={handleOpen}>{'${texts.RANKING_BUTTON}'}</Button>
184         </div>
185
186         <Modal
187           open={open}
188           onClose={handleClose}
189           aria-labelledby="parent-modal-title"
190           aria-describedby="parent-modal-description"
191           style={{ display: 'flex', alignItems: 'center', justifyContent: 'center' }}
192         >
193           <Card id='ranking-card' className='unselectable'>
194             <Scrollbar>
195               <CardContent>
196                 {promotion != null
197                   ?
198                     <div className='ranking-modal'>
199                       <Grid container spacing={4} direction='column' style={styles.gridContainer}>
200
201                         <Grid item container direction='row' style={{ ...styles.gridContainer, marginTop: '30px', height: '5vh' }}>
202                           <Grid item container style={{ justifyContent: 'space-between' }}>
203                             <Grid item>
204                               <IconButton aria-label="Volver" sx={{ color: '#FFFFFF' }} onClick={handleClose}>
205                                 <ArrowBackIcon />
206                               </IconButton>
207                             </Grid item>
208                           </Grid>
209                         </Grid item>
210                       </Grid>
211
212                       <Grid item style={{ ...styles.gridContainer, textAlign: 'center' }}>
213                         <Typography id='ranking-title' style={styles.textTitle}>{'${texts.TITLE_OF_RANKING}'}</Typography>
214                       </Grid>
215
216                       <Grid item style={{ ...styles.gridContainer, textAlign: 'center' }}>
217                         <Typography id='ranking-subtitle' style={styles.textSubTitle}>{'${texts.SUBTITLE_OF_RANKING}'}</Typography>
218                       </Grid>
219                     </div>

```

```

221             <Grid item container style={{ ...styles.gridContainer }}>
222               <Card id='ranking-form-card'>
223                 <CardContent>
224                   <Formik
225                     initialValues={{ email: '' }}
226                     onSubmit={async (values, helpers) => {
227                       const userRanking = await request(values.email)
228                       if (userRanking) {
229                         setUserPositionRanking(userRanking)
230                       } else {
231                         setUserPositionRanking(0)
232                       }
233                       helpers.resetForm()
234                     }}
235                     validationSchema={Yup.object({
236                       email: Yup.string().matches(/^[a-z0-9!#$%&'*/~?^_{}|]+(?:\.[a-z0-9!#$%&'*/~?^_{}|]+)*@(?:(?![a-z0-9]{4,})(?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\.)+)+$/i, '$(texts.EMAIL_VAL)').required('$(texts.REQUIRED_TEXT_VAL)')
237                     })}
238                   >
239                     <Form autoComplete="off">
240                       <Box paddingBottom={2}>
241                         <Field type='email' fullWidth name='email' component={TextField} label={texts.FORM_LABEL_EMAIL} variant="outlined"
242                         InputLabelProps={{ id: 'form-label-email', style: styles.textField }} />
243                       </Box>
244                       <Grid container spacing={2} style={styles.gridContainer}>
245                         <Grid item>
246                           <Button
247                             disabled={isSubmitting} variant='contained'
248                             id='ranking-button-form'
249                             startIcon={isSubmitting ? <CircularProgress size='1rem' /> : null}
250                             style={{ backgroundColor: isSubmitting ? '#C0C0C0' : '#140289', color: isSubmitting ? '#757575' : '#FFFFFF' }}
251                             type='submit'
252                             {'${texts.SEE_POSITION_BUTTON}'}
253                           </Button>
254                         </Grid>
255                       </Grid>
256                     </Form>
257                   </Formik>
258                 </CardContent>
259               </Card>
260             </Grid item>

```

```

262         {isCompleted
263         ? userPositionRanking > 0
264         :
265         <Grid item style={({ ...styles.gridContainer, marginTop: '50px', textAlign: 'center' })}>
266           <Typography id='desc-part-pos' style={({ ...styles.textSubTitle })>{texts.DESCRPTION_OF_THE_RANKING_POSITION}</Typography>
267           <Typography id='pos-part' style={({ ...styles.textSubTitle, ...styles.textBorder })>{userPositionRanking}</Typography>
268           <Typography id='desc-part-price' style={styles.textSubTitle}>{texts.DESCRPTION_OF_THE_PRIZE_IN_RANKING}</Typography>
269           <Typography id='part-prize-price-3' style={({ ...styles.textSubTitle, ...styles.textBorder })>{userPositionRanking <= promotion.
270             firstPlaces ? promotion.prizeMaxPrice : promotion.prizeMinPrice}</Typography>
271         </Grid>
272         : <Grid item style={({ ...styles.gridContainer, marginTop: '30px', textAlign: 'center' })}>
273           <Typography id='email-not-found' style={({ ...styles.textSubTitle, marginBottom: '10px', fontSize: '18px' })>{texts.EMAIL_NOT_FOUND}</Typography>
274           <Typography>
275             {promotion.expired === false && promotion.couponsAvailabes
276               ? <Typography id='desc-part-without-pos' style={styles.textSubTitle}>{texts.INVITATION_TO_PARTICIPATE}</Typography>
277               : null
278             }
279         </Grid>
280         : null
281       )
282     </Grid>
283     <ChildModal />
284   </Grid>
285 </div>
286 : null
287 </div>
288 : null
289 </div>
290 : null
291 </div>
292 </CardContent>
293 </Scrollbar>
294 </Card>
295 </Modal>
296 </div>
297 )
298 }

```

El componente Ranking retorna un formulario de consulta de posición mediante un email ya registrado. Si el Usuario que consulta mediante su email está registrado en la campaña, se le muestra su posición y premio a obtener, en cambio si no lo está se mostrará un mensaje de email no encontrado.

Ranking vendrá acompañado de componentes tales como MaterialUI, ScrollBar Custom y Formik que le darán estilo y funcionalidad:

- **Button:** Como su nombre lo indica, un botón que permite al Usuario acceder a este mismo componente Modal.
- **Modal:** componente que proporciona una base sólida para ventanas pop-up. El mismo cuenta con la propiedad “open” que utiliza un estado booleano para saber cuándo abrirse y “onClose” que es un evento que junto a handleClose() permiten al Usuario salir del Modal tocando todo lo externo a él, es decir, su fondo. Además handleClose(), al setear el ‘isCompleted’ en false, elimina el último mensaje mostrado al Usuario respecto a su posición.

- **Card:** Superficie que muestra contenido sobre un solo tema.
- **Scrollbar:** Como su nombre lo indica, una barra de scroll que permite deslizarse cuando las pantallas son reducidas y el contenido debe ajustarse.
- **CardContent:** Permite mostrar el contenido deseado.
- **Grid:** Cuadrícula que permite flexibilidad en una amplia variedad de diseños.
- **IconButton:** componente que permite (junto a ArrowBackIcon) representar un ícono para 'volver'.
- **Typography:** componente que representa contenido textual.
- **Formik:** componente encargado del completo manejo del formulario, es decir: el manejo de envío del mismo, el manejo de los valores/estados y de las validaciones y mensajes de error (junto a otro Componente llamado "Yup")
- **Box:** Componente contenedor.
- **Field:** Conecta automáticamente las entradas del Usuario a Formik. Utiliza el atributo 'name' para que coincida con el estado inicial de Formik y así poder asignar cada valor a la propiedad indicada. A su vez, como vemos, en sus propiedades contiene:
 - **initialValues:** el mismo contiene el valor inicial de la propiedad a utilizar en el formulario (email), gracias a este objeto, Formik puede asignar el registro que el Usuario haya ingresado.
 - **onSubmit():** función que permite el manejo del envío del formulario:
 - **request():** función que hace una petición para obtener la posición solicitada del Usuario, la misma utiliza el email ingresado por el Usuario. Si el email ingresado estaba registrado previamente, el mismo se retornará para su uso posterior, pero si el email ingresado no estaba registrado se retornará un null que posteriormente será considerado en las validaciones. En cualquiera de los dos casos posibles, se seteará el estado isCompleted con setIsCompleted para

luego usarlo como validación de muestra de mensajes al Usuario, ya que el mismo es el que nos dice si la petición pudo realizarse.

- **setUserPositionRanking():** función que sirve para setear el estado actual del Usuario, si la posición retornada es numérica se usará ese valor al momento de setear, en cambio si la posición retornada no es numérica se seteará el estado con un 0 (cero).
- **validationSchema:** junto a Yup, se hace la siguiente validación al campo del formulario:
 - **email:**

Para finalizar, el componente Ranking, además de retornar el componente Formik, como es debido, retorna el formulario que el Usuario utilizara para su consulta sobre su posición y premio actual. El mismo tendrá:

- Un campo para el ingreso del email.
- Un botón: para continuar, y realizar la búsqueda.

... Y los respectivos mensajes de aviso al Usuario de su participación junto a la posición y premio a obtener en la campaña, en el caso de estar registrado; y en el caso de que no esté registrado el mensaje “email no fue encontrado” sumado a una invitación a participar.

Finalmente, el componente Ranking, tendrá los siguientes estilos que serán usados a lo largo de todo el componente:

```
303   const styles = {
304     gridContainer: {
305       justifyItems: 'center',
306       alignItems: 'center',
307       justifyContent: 'center',
308       alignContent: 'center'
309     },
310     textTitle: {
311       color: 'FFFFFF',
312       fontSize: '25px'
313     },
314     textSubTitle: {
315       color: 'FFFFFF',
316       fontSize: '20px',
317     },
318     textDescription: {
319       color: 'FFFFFF'
320     },
321     textField: {
322       fontSize: 14
323     },
324     textBorder: {
325       borderRadius: '50px',
326       borderColor: '#14D2B9',
327       borderWidth: '3px',
328       borderStyle: 'solid',
329       width: '50%',
330       margin: '10px auto'
331     }
332   }
```

CouponMessage

El componente CouponMessage es utilizado, luego del registro en la campaña vigente, para mostrarle al Usuario en una ventana Modal tipo pop-up el cupón de descuento obtenido y la URL de referido con la cual puede ir sumando puntos en el Ranking.

A continuación mostraremos y explicaremos el código empleado:

- **Importaciones:**

```
1 import React, {useState, useContext} from "react"
2 import { texts } from '../assets/texts/strings.js'
3 import { Card, CardContent, Button, Grid, Modal, Typography } from '@material-ui/core'
4 import { Scrollbar } from 'react-scrollbars-custom'
5 import { CopyToClipboard } from 'react-copy-to-clipboard';
6 import ArrowBackIcon from '@mui/icons-material/ArrowBack';
7 import IconButton from '@mui/material/IconButton';
8 import FormContext from '../context/Form/FormContext.js'
9 import PromotionContext from '../context/Promotion/PromotionContext.js'
```

Podemos apreciar que importamos:

- El hook de **'useState'**: para poder hacer uso de un estado en un componente de función.
- El hook de **'useContext'**: para poder acceder y utilizar variables globales.
- Objeto **texts**: para acceder a los textos necesarios para cada componente.
- Componentes de **MaterialUI**: para poder aplicar estilos.
- Librería **react-scrollbars-custom**: para poder agregar una barra deslizable al componente.
- Librería **react-copy-to-clipboard**: como su nombre lo indica, para poder agregar la funcionalidad de copiar al portapapeles.
- Icono **'ArrowBackIcon'** para poder volver atrás en cada modal.
- Componente de contexto **'FormContext'**: para también poder acceder a las variables globales del formulario.
- Componente de contexto **'PromotionContext'**: para también poder acceder a las variables globales de la promoción vigente.

- **Componente CouponMessage:**

```
11 export const CouponMessage = () => {  
12   const { promotion } = useContext(PromotionContext);  
13   const { coupon, openCouponModal, setOpenCouponModal } = useContext(FormContext);  
14   const [copyTextLink, setCopyTextLink] = useState({ copied: false });  
15   const [copyTextCoupon, setCopyTextCoupon] = useState({ copied: false });  
16   const handleClose = () => setOpenCouponModal(false)
```

Podemos apreciar:

- El uso del objeto global **promotion**, que va a ser utilizado durante todo el componente para el acceso a sus propiedades que describen toda la promoción vigente, es decir, los precios que maneja, las descripciones de la misma, las condiciones que maneja, etc.
- El uso de las **variables y funciones globales de Formulario** (en este caso relacionadas al cupón que el Usuario ganó), las mismas van a ser utilizadas durante todo el componente para el acceso a sus funcionalidades y propiedades:
 - Variable booleana **openCouponModal** y función **setOpenCouponModal**: la primera le permite saber al Modal, utilizado en este componente, cuando abrirse y cuando cerrarse ya que la misma es pasada a su propiedad 'open', la segunda permite hacer el manejo de cerrado del modal mediante un botón de 'volver' o clickeando en cualquier parte externa al modal, es decir, su fondo.
 - Objeto **coupon**: el mismo tendrá la URL de referido la cual el Usuario puede compartir a sus contactos para poder sumar puntos en el Ranking vigente, y además el cupón textual en sí, que aplicará en sus transacciones.

- Un hook de estado booleano '**copyTextLink**' para indicar cuando el Usuario realizó una copia de la URL de referido o no.
- Un hook de estado booleano '**copyCouponLink**' para indicar cuando el Usuario realizó una copia del cupón obtenido o no.
- Función **handleClose**, la misma es usada en el componente Modal para saber qué hacer en su evento onClose, en este caso, setear la variable global booleana **openCouponModal** para así poder cerrar el Modal.

Finalmente el componente CouponMessage **retorna**:

```

19   return (
20     <Modal
21       open={openCouponModal}
22       onClose={handleClose}
23       style={{ display: 'flex', alignItems: 'center', justifyContent: 'center' }}
24     >
25       <Card id="message-coupon-card">
26         <Scrollbar>
27           <CardContent>
28             <Grid container spacing={3} direction="column" style={{ ...styles.gridContainer, minHeight: '55vh' }}>
29               <Grid item container direction="row" style={{ ...styles.gridContainer, marginTop: '30px' }}>
30                 <Grid item container style={{ justifyContent: 'space-between' }}>
31                   <Grid item>
32                     <IconButton aria-label="Volver" sx={{ color: '#FFFFFF' }} onClick={handleClose}>
33                       <ArrowBackIcon />
34                     </IconButton>
35                   </Grid>
36                 </Grid>
37             </Grid>
38             <Grid>
39               <Grid item container direction="column" style={{ ...styles.gridContainer, textAlign: 'center', margin: '0 auto' }}>
40                 <Typography id="coupon-desc" style={{ color: '#FFFFFF', fontSize: '22px', margin: '5px auto' }}>{promotion.coupon.longDescription}</Typography>
41                 <Typography id="coupon-desc-2" style={{ color: '#FFFFFF', fontSize: '20px', margin: '5px auto' }}>{promotion.coupon.pricingShortDescription}</Typography>
42               </Grid>
43               <Grid item style={styles.borderCoupon}>
44                 <Typography id="coupon" style={{ color: '#FFFFFF', fontSize: '20px' }}>{coupon.coupon}</Typography>
45               </Grid>
46               <Typography id="desc-referr-link" style={{ color: '#FFFFFF', fontSize: '18px', margin: '20px 0px' }}>{promotion.descriptionSharePrizeCoupon}</Typography>
47               <Grid item style={styles.borderCoupon}>
48                 <Typography id="referr-link" style={{ color: '#FFFFFF', fontSize: '18px' }}>
49                   <a href={coupon.url_referrals} target="_blank" style={{ color: '#FFFFFF' }}>{coupon.url_referrals}</a>
50                 </Typography>
51               </Grid>
52             </Grid>
53           </CardContent>
54         </Scrollbar>
55       </Card>
56     </Modal>
57   )
58
59   <Grid item container direction="row" style={{ ...styles.gridContainer, textAlign: 'center', width: '50%', marginTop: '30px' }}>
60     <Grid item style={{ ...styles.gridContainer, textAlign: 'center', margin: '15px auto' }}>
61       <CopyToClipboard>
62         <Text>{coupon.coupon}</Text>
63         <onCopy={() => setCopyTextCoupon({ ...copyTextCoupon, copied: true })}</onCopy>
64       </CopyToClipboard>
65       <Button
66         id="coupon-button-1"
67         variant="contained"
68         style={styles.buttonCoupon}
69       >{texts.COUPON_MC_BUTTON}</Button>
70     </Grid>
71     {copyTextCoupon.copied ? <Typography id="copy-coupon" style={styles.copiedText}>{texts.TEXT_COPIED}</Typography> : null}
72   </Grid>
73
74   <Grid item style={{ ...styles.gridContainer, textAlign: 'center', margin: '15px auto' }}>
75     <CopyToClipboard>
76       <Text>{coupon.url_referrals}</Text>
77       <onCopy={() => setCopyTextLink({ ...copyTextLink, copied: true })}</onCopy>
78     </CopyToClipboard>
79     <Button
80       id="coupon-button-2"
81       variant="contained"
82       style={styles.buttonCoupon}
83     >{texts.LINK_MC_BUTTON}</Button>
84   </Grid>
85   {copyTextLink.copied ? <Typography id="copy-link" style={styles.copiedText}>{texts.TEXT_COPIED}</Typography> : null}
86 </Grid>
87 </CardContent>
88 </Scrollbar>
89 </Card>
90 </Modal>
91 )
92

```

El componente CouponMessage retorna una ventana Modal tipo pop-up en la cual se lo felicita al Usuario por haber obtenido el cupón de la campaña vigente, la misma le indica al Usuario el descuento que obtuvo con el cupón ganado, el tope de reintegro del mismo, el código del cupón (por ejemplo, AYSA20211), la URL de referidos a compartir y además la

posibilidad de copiar estas dos últimas mediante botones que hacen la copia automáticamente.

CouponMessage vendrá acompañado de componentes tales como MaterialUI, ScrollBar Custom y CopyToClipboard que le darán estilo y funcionalidad:

- **Modal:** componente que proporciona una base sólida para ventanas pop-up. El mismo cuenta con la propiedad “open” que utiliza un estado booleano para saber cuándo abrirse y “onClose” que es un evento que junto a handleClose() permiten al Usuario salir del Modal tocando todo lo externo a él, es decir, su fondo.
- **Card:** Superficie que muestra contenido sobre un solo tema.
- **Scrollbar:** Como su nombre lo indica, una barra de scroll que permite deslizarse cuando las pantallas son reducidas y el contenido debe ajustarse.
- **CardContent:** Permite mostrar el contenido deseado.
- **Grid:** Cuadrícula que permite flexibilidad en una amplia variedad de diseños.
- **IconButton:** componente que permite (junto a ArrowBackIcon) representar un ícono para ‘volver’.
- **Typography:** componente que representa contenido textual.
- **CopyToClipboard:** componente encargado de, como su nombre lo indica, hacer copia al portapapeles del Usuario. El mismo tendrá dos propiedades, ‘text’ que viene a ser el texto a copiar y ‘onCopy’ que viene a ser el callback que se llame cuando se copie el texto (dentro usará la función setCopyTextCoupon o setCopyTextLink, ya que este componente se utilizará dos veces.)
- **Button:** Como su nombre lo indica, un botón que en este caso permitirá al Usuario copiar la URL de referido o el cupón obtenido.

Finalmente, el componente CouponMessage, tendrá los siguientes estilos que serán usados a lo largo de todo el componente:

```
94  const styles = {
95    gridContainer: {
96      justifyItems: 'center',
97      alignItems: 'center',
98      justifyContent: 'center',
99      alignContent: 'center'
100   },
101   borderCoupon: {
102     borderRadius: '50px',
103     borderColor: '#14D2B9',
104     borderWidth: '3px',
105     borderStyle: 'solid',
106     margin: '10px auto',
107     padding: '10px'
108   },
109   buttonCoupon: {
110     backgroundColor: '#14D2B9',
111     color: 'FFFFFF'
112   },
113   copiedText: {
114     color: 'FFFFFF',
115     fontSize: '13px',
116     margin: '5px auto'
117   },
118   textField: {
119     fontSize: '14px'
120   }
121 }
```

TermsMessage

El componente TermsMessage es utilizado, en el momento del registro en la campaña vigente, para mostrarle al Usuario en una ventana Modal tipo pop-up los términos y condiciones.

A continuación mostraremos y explicaremos el código empleado:

- **Importaciones:**

```
1 import React, { useContext } from "react"
2 import { texts } from '../assets/texts/strings.js'
3 import { Card, CardContent, Grid, Modal, Typography } from '@material-ui/core'
4 import { Scrollbar } from 'react-scrollbar-custom'
5 import ArrowBackIcon from '@mui/icons-material/ArrowBack';
6 import IconButton from '@mui/material/IconButton';
7 import FormContext from '../context/Form/FormContext.js'
8 import PromotionContext from "../context/Promotion/PromotionContext.js";
```

Podemos apreciar que importamos:

- El hook de **'useContext'**: para poder acceder y utilizar variables globales.
- Objeto **texts**: para acceder a los textos necesarios para cada componente.
- Componentes de **MaterialUI**: para poder aplicar estilos.
- Librería **react-scrollbar-custom**: para poder agregar una barra deslizable al componente.
- Icono **'ArrowBackIcon'** para poder volver atrás en cada modal.
- Componente de contexto **'FormContext'**: para también poder acceder a las variables globales del formulario.
- Componente de contexto **'PromotionContext'**: para también poder acceder a las variables globales de la promoción vigente.

- **Componente TermsMessage:**

```
10 export const TermsMessage = () => {  
11   const { openTermsModal, setOpenTermsModal } = useContext(FormContext);  
12   const { promotion } = useContext(PromotionContext);  
13   const handleCloseTerms = () => setOpenTermsModal(false)  
14 }
```

Podemos apreciar:

- El uso del objeto global **promotion**, que va a ser utilizado durante todo el componente para el acceso a sus propiedades que describen toda la promoción vigente, es decir, los precios que maneja, las descripciones de la misma, las condiciones que maneja, etc.
- El uso de la **variable y función global de Formulario** referidas al manejo del Modal de términos y condiciones:
 - Variable booleana **openTermsModal** y función **setOpenTermsModal**: la primera le permite saber al Modal, utilizado en este componente, cuando abrirse y cuando cerrarse ya que la misma es pasada a su propiedad '**open**', la segunda permite hacer el manejo de cerrado del modal mediante un botón de 'volver' o clickeando en cualquier parte externa al modal, es decir, su fondo.
- Función **handleCloseTerms**, la misma es usada en el componente Modal para saber qué hacer en su evento onClose, en este caso, setear la variable global booleana **openTermsModal** para así poder cerrar el Modal.

El componente TermsMessage **retorna**:

```

16  return (
17    <Modal
18      open={openTermsModal}
19      onClose={handleCloseTerms}
20      style={{ display: 'flex', alignItems: 'center', justifyContent: 'center' }}
21    >
22      <Card id='message-terms-card'>
23        <Scrollbar>
24          <CardContent>
25            <Grid container spacing={4} direction="column" style={{ ...styles.gridContainer }}>
26              <Grid item container direction="row" style={{ ...styles.gridContainer, marginTop: '30px', height: '5vh' }}>
27                <Grid item container style={{ justifyContent: 'space-between' }}>
28                  <Grid item>
29                    <IconButton aria-label="Volver" sx={{ color: '#FFFFFF' }} onClick={handleCloseTerms}>
30                      <ArrowBackIcon />
31                    </IconButton>
32                  </Grid>
33                </Grid>
34              </Grid>
35            </Grid>
36            <Grid item style={{ ...styles.gridContainer, textAlign: 'center' }}>
37              <Typography id='terms-title' style={styles.textTitle}>{`${texts.TITLE_OF_TERMS`}</Typography>
38            </Grid>
39            {promotion && promotion.expired === false && promotion.couponsAvailables
40              ?
41                <Grid item container direction="column" style={{ ...styles.gridContainer, textAlign: 'center', margin: '0 auto' }}>
42                  <Typography id='terms-desc' style={{ color: '#FFFFFF', fontSize: '1rem', margin: '5px auto', whiteSpace: 'pre-line' }}>{promotion.
43                    descriptionTermsAndConditions}</Typography>
44                </Grid>
45              : null
46            }
47          </Grid>
48        </CardContent>
49      </Scrollbar>
50    </Card>
51  </Modal>
52  )
53  )
54  )
55  )

```

El componente TermsMessage retorna una ventana Modal tipo pop-up, si la promoción sigue vigente y hay cupones disponibles, en la cual se le muestra al Usuario los Términos y Condiciones.

TermsMessage vendrá acompañado de componentes tales como MaterialUI y ScrollBar Custom que le darán estilo y funcionalidad:

- **Modal:** componente que proporciona una base sólida para ventanas pop-up. El mismo cuenta con la propiedad “open” que utiliza un estado booleano para saber cuándo abrirse y “onClose” que es un evento que junto a handleCloseTerms() permiten al Usuario salir del Modal tocando todo lo externo a él, es decir, su fondo.

- **Card:** Superficie que muestra contenido sobre un solo tema.
- **Scrollbar:** Como su nombre lo indica, una barra de scroll que permite deslizarse cuando las pantallas son reducidas y el contenido debe ajustarse.
- **CardContent:** Permite mostrar el contenido deseado.
- **Grid:** Cuadrícula que permite flexibilidad en una amplia variedad de diseños.
- **IconButton:** componente que permite (junto a ArrowBackIcon) representar un ícono para 'volver'.
- **Typography:** componente que representa contenido textual.

Finalmente, el componente TermsMessage, tendrá los siguientes estilos que serán usados a lo largo de todo el componente:

```
57  const styles = {
58    gridContainer: {
59      justifyItems: 'center',
60      alignItems: 'center',
61      justifyContent: 'center',
62      alignContent: 'center'
63    },
64    borderCoupon: {
65      borderRadius: '50px',
66      borderColor: '#14D2B9',
67      borderWidth: '3px',
68      borderStyle: 'solid',
69      margin: '10px auto',
70      padding: '10px'
71    },
72  },
```



```
72     buttonCoupon: {  
73       backgroundColor: '#14D2B9',  
74       color: 'FFFFFF'  
75     },  
76     copiedText: {  
77       color: 'FFFFFF',  
78       fontSize: '13px',  
79       margin: '5px auto'  
80     },  
81     textField: {  
82       fontSize: '14px'  
83     },  
84     textTitle: {  
85       color: 'FFFFFF',  
86       fontSize: '25px'  
87     }  
88   }
```

App

El componente App es utilizado como el esqueleto principal de la Landing Web, ya que el mismo contiene la pantalla principal más todos los componentes que componen al proyecto.

A continuación mostraremos y explicaremos el código empleado:

- **Importaciones:**

```
1 import React, { useEffect, useState, useContext } from 'react'
2 import './App.css';
3 import { texts } from './assets/texts/strings.js'
4 import logo from './assets/images/TAP_marca-02-color-RGB-gradiente-invertido.png'
5 import FormState from './context/Form/FormState.js';
6 import PromotionState from './context/Promotion/PromotionState.js';
7 import PromotionContext from './context/Promotion/PromotionContext';
8 import FormContext from './context/Form/FormContext';
9 import { CouponMessage } from './components/CouponMessage.js';
10 import Formulary from './components/Formulary.js';
11 import Ranking from './components/Ranking.js'
12 import { TermsMessage } from './components/TermsMessage.js';
13 import { getPromotion } from './strapi/data.js'
14 import { getApiURL } from './strapi/config.js';
15 import { BrowserRouter as Router, Route } from "react-router-dom";
16 import swal from 'sweetalert';
17 import { useSpring, a, config } from '@react-spring/web'
18 import { useDrag } from '@use-gesture/react'
```

- Podemos apreciar que importamos:
 - El hook de **'useEffect'**: para poder realizar operaciones secundarias que impliquen, por ejemplo, una petición de datos.
 - El hook de **'useState'**: para poder hacer uso de un estado en un componente de función.
 - El hook de **'useContext'**: para poder acceder a variables globales.
 - Objeto **texts**: para acceder a los textos necesarios para cada componente.
 - Componente de estado **'FormState'**: contiene el estado global de Formulario más las funciones que lo manejan. Es utilizado en este caso para poder actuar como wrapper de los demás componentes presentes en todo el proyecto, de este modo ellos podrán acceder a sus propiedades un hook de contexto.

- Componente de estado '**PromotionState**': contiene el estado global de Promoción más las funciones que lo manejan. Es utilizado en este caso para poder actuar como wrapper de los demás componentes presentes en todo el proyecto, de este modo ellos podrán acceder a sus propiedades usando un hook de contexto.
- Componente de contexto '**PromotionContext**': para también poder acceder a las variables globales de la promoción vigente.
- Componente de contexto '**FormContext**': para también poder acceder a las variables globales del formulario.
- Componente **CouponMessage**: para poder mostrar la ventana Modal de tipo pop-up que visualiza el cupón obtenido.
- Componente **Formulary**: para poder utilizar y visualizar las funciones del formulario de registro a la campaña vigente.
- Componente **Ranking**: para poder visualizar la lista de participantes que forman parte de la promoción vigente. A su vez, los Usuarios también podrán consultar su posición actual y el premio obtenido al momento.
- Componente **TermsMessage**: para poder mostrar la ventana Modal de tipo pop-up que visualiza los términos y condiciones.
- **Logo**: imagen .png del logo de TAP, utilizada en el header de la Landing Web.
- Función **getPromotion()**: para poder obtener la promoción vigente.
- Función **getApiURL**: para poder obtener la dirección web de donde está alojado Strapi.
- Librería **sweetalert**: para poder realizar las alertas de error correspondientes.
- Librería '**@react-spring/web**': es una librería de animación que cubre la mayoría de animaciones relacionadas con la interfaz de usuario. De ella usaremos 'a' (para agregar delante de cada componente utilizado) y

‘useSpring’ para convertir los valores en valores animados, y además ‘config’ para poder configurar estos efectos que Spring nos proporciona.

- Librería ‘@use-gesture/react’: es un conjunto de gestos que permiten vincular eventos táctiles y de mouse a cualquier nodo.
- Librería ‘react-router-dom’: es una librería para gestionar rutas en aplicaciones que utilicen ReactJS.

- **Componente App:**

```
21 function App() {
22   const { promotion, setPromotion } = useContext(PromotionContext);
23   const { openCouponModal, openTermsModal } = useContext(FormContext);
24   const [isClosed, setIsClosed] = useState(true);
25   const [y, api] = useSpring(() => ({ y: 0 }))
26   const maxHeight = -(window.innerHeight / 2 + 60);
27
28   const open = ({ canceled }) => {
29     api.start({ y: maxHeight, immediate: false, config: canceled ? config.wobbly : config.stiff })
30     setIsClosed(false)
31   }
32
33   const close = (velocity = 0) => {
34     api.start({ y: 0, immediate: true, config: { ...config.stiff, velocity } })
35     setIsClosed(true)
36   }
37
```

```
37
38   const bind = useDrag(
39     ({ last, velocity: [, vy], direction: [, dy], offset: [, oy], cancel, canceled }) => {
40       if (oy < -550 || oy > 100) {
41         cancel()
42       }
43
44       if (last) {
45         oy > -200 || (vy > 0.5 && dy > 0) ? close(vy) : open({ canceled })
46       } else {
47         api.start({ y: oy, immediate: true })
48       }
49     },
50     { from: () => [0, y.get()], filterTaps: true, bounds: { top: maxHeight }, rubberband: true }
51   )
52
53   const formatDate = dateString => {
54     let date = new Date(dateString)
55     date = new Date(date.getFullYear(), date.getMonth(), date.getDate() + 1)
56     return date.toLocaleDateString('en-GB')
57   }
```

```
59   useEffect(() => {
60
61     async function fetchMyAPI() {
62       await getPromotion()
63       .then(data => setPromotion(data))
64       .catch(() => {
65         swal({
66           title: "¡Error!",
67           text: `${texts.ERROR_NO_CURRENT_PROMOTION}`,
68           icon: 'error',
69           button: {
70             text: "Aceptar",
71           },
72           timer: 10000
73         });
74       })
75     }
76
77     fetchMyAPI()
78
79   }, []); // eslint-disable-line react-hooks/exhaustive-deps
```

Podemos apreciar:

- El uso del objeto global **promotion**, que va a ser utilizado durante todo el componente para el acceso a sus propiedades que describen toda la promoción vigente, es decir, los precios que maneja, las descripciones, las condiciones que maneja, etc. Además, en este componente setearemos la promoción vigente luego de hacer una petición de la misma.
- El uso de las **variables globales de Formulario** (en este caso relacionadas a las ventanas Modal de tipo pop-up), las mismas van a ser utilizadas cuando el Usuario termine con su registración de forma exitosa y cuando desee leer los términos y condiciones:
 - Variable booleana **openCouponModal** y **openTermsModal**: cada una de ellas le permite saber al Modal correspondiente, utilizado en

este proyecto, cuando abrirse y cuando cerrarse ya que la misma es pasada a su propiedad **'open'**.

- Un hook de estado booleano **'isClosed'** para que cuando el Usuario clickee sobre el pseudoboton del formulario se abra o se cierre dependiendo del estado actual.
- El uso de **useSpring**, que nos devuelve la posición del 'eje y' y el objeto 'api' con el cual haremos actualizaciones de animación al componente que utilice Spring.
- **Constante numérica** que nos devuelve la mitad de la altura de la ventana web.

- **Función open:**

Recibe una propiedad **'canceled'** que indica si el gesto fue cancelado (arrastrar o pellizcar) y se encarga de hacer uso del objeto api para actualizar el estado de la animación e iniciarla (**api.start()**) con los siguientes valores:

- **y:** esta propiedad trata de la posición en el 'eje y', la misma es seteada con la constante numérica de la mitad de la altura de la ventana web.
- **immediate:** esta propiedad es seteada en false debido a que evita la animación si es seteada en true.
- **config:** esta propiedad, como se explicó anteriormente, nos permite configurar los efectos que Spring nos proporciona. Si canceled es true significa que fue cancelado y por ende se hará uso de 'config.woobly' que es una de las configuraciones por defecto que Spring nos proporciona ({ mass: 1, tension: 180, friction: 12 }), en cambio sí es false significa que no fue cancelado y por ende se hará uso de 'config.stiff' que es otra de las

configuraciones por defecto que Spring nos proporciona ({ mass: 1, tension: 210, friction: 20 }).

Además se encarga de setear la variable de estado booleana **isClosed** en false para que cuando el Usuario presione el pseudoboton del formulario se ejecute la función de cierre.

- **Función close:**

La misma recibe una **velocidad** con la cual se realizará el gesto (por defecto será 0) y se encarga de hacer uso del objeto api para actualizar el estado de la animación e iniciarla (**api.start()**) con los siguientes valores:

- **y:** esta propiedad trata de la posición en el 'eje y', la misma es seteada en 0.
- **immediate:** esta propiedad es seteada en true debido a que no evita la animación si es seteada en false.
- **config:** esta propiedad, como se explicó anteriormente, nos permite configurar los efectos que Spring nos proporciona. Se hará uso de 'config.stiff' que es otra de las configuraciones por defecto que Spring nos proporciona ({ mass: 1, tension: 210, friction: 20 }) y además se le agregará a esta configuración la velocidad recibida por argumento.

Además, se encarga de setear la variable de estado booleana **isClosed** en true para que cuando el Usuario presione el pseudoboton del formulario se ejecute la función de apertura.

- **Constante bind:**

- La misma contiene el hook **useDrag** el cual devuelve una función, que cuando es llamada retorna un objeto con controladores de eventos. Esta constante es utilizada para extender con `{... bind ()}` en un componente, lo cual internamente agrega controladores de eventos `onMouseDown` y `onTouchStart`.
- **useDrag** no es responsable de mover el componente, solo entrega datos de gestos a `react-spring` que establece las transformaciones de componentes.

Para hacer uso de **useDrag**, debemos pasarle:

- **Función Callback**, que recibe como parámetro:
 - Un objeto **'state'**, que contiene todos los atributos del gesto, incluido el evento original. Ese estado se pasa a su controlador cada vez que se actualiza el gesto.
 - Internamente la función hará uso de los atributos recibidos.
- **Config**: Un objeto el cual contiene opciones para el gesto a usar.

En el caso del componente **App.js**, la constante **bind** utilizada para vincularla a un componente (`<div>` contenedor del Formulario), consta de las siguientes características:

- **Función callback**, que recibe por argumento el objeto `state` con los atributos:
- **last**: variable booleana que es `'true'` cuando es el último evento.

- **velocity:** variable numérica que representa el impulso del gesto por eje. Hacemos uso de la velocidad en el eje y.
- **direction:** variable numérica que representa la dirección por eje. Hacemos uso de la dirección en el eje y
- **offset:** variable numérica que representa el desplazamiento/distancia en coordenadas desde la posición del primer gesto. Hacemos uso del offset en el eje y de la coordenada resultante.
- **cancel:** función a la que se puede llamar para interrumpir algunos gestos.
- **canceled:** variable booleana que nos indica si el gesto fue cancelado.

Internamente esta función hace:

- Si el **offset del eje y** del componente arrastrado al finalizar el arrastre nos da que es menor a -550 o mayor a 100 se hará uso de la función **cancel** la cual interrumpe los posibles gestos y devuelve el componente a su posición original.
- Si finalizó el arrastre del componente, es decir **last** nos indica que ya es el último evento, entonces hacemos la siguiente consulta:
 - Si el **offset del 'eje y'** queda mayor a -200 o su velocidad en el **'eje y'** queda mayor a 0.5 y su dirección en el **'eje y'** queda mayor a 0, el componente hace uso de la función **'close()'** la cual le pasa la velocidad resultante; en cambio si el resultado no es ninguno de esas dos consideraciones, el componente hace uso de la función **'open()'** la cual le pasa la variable booleana que nos indica si al finalizar el gesto el mismo fue cancelado o no.
- Si el arrastre del componente no finalizó, es decir, **last** nos indica que aún no es el último evento, entonces hacemos lo siguiente: hacemos uso del objeto

api para actualizar el estado de la animación e iniciarla (**api.start()**) con los siguientes valores:

- **y:** esta propiedad trata de la posición en el 'eje y', la misma es seteada con 'oy', es decir con la variable numérica 'offset y' que representa el desplazamiento/distancia en coordenadas desde la posición del primer gesto.
- **immediate:** esta propiedad es seteada en true para evitar la animación.

- **Config:**

Como segundo parámetro pasamos el objeto config con las siguientes propiedades que personalizan el gesto a usar:

- **from:** esta propiedad, recibe una función que retorna un array con el eje de coordenadas X e Y. Si nosotros establecemos que cuando el Usuario arrastre el formulario pueda moverlo cuando el mismo esté bajando o subiendo y eso no afecte a su trayecto, es decir, si el Usuario quiere subir el formulario mientras el mismo está bajando, nosotros queremos que el formulario permita ese flujo. Por lo tanto, para lograr eso, debemos establecer que el valor offset del eje Y se establezca de acuerdo a los valores que el Usuario genere con su arrastre. (No usamos el eje X porque el formulario solo se mueve en vertical)
 - **El valor en el eje Y** que establecemos es el visto en código '**y.get()**', el mismo nos permite que el flujo del formulario no se vea afectado por las intervenciones del Usuario, sino que se adapte a los valores que él genera.

- **filterTaps:** esta propiedad booleana pasada en 'true' es debido a que la misma nos permite que si es verdadera, el componente no activará su lógica de arrastre si el Usuario simplemente hizo clic en el componente.
 - **bounds:** esta propiedad de tipo objeto nos permite indicar un límite en el desplazamiento del gesto. Por lo tanto, hemos puesto que el formulario solo pueda ser elevado a una altura máxima de la mitad de la ventana web.
 - **rubberband:** esta propiedad booleana nos permite establecer el coeficiente de elasticidad del gesto al salir de los límites indicados. Cuando se establece en verdadero, el coeficiente de elasticidad se establecerá por defecto en 0,15.
-
- Función **formatDate()**: recibe como argumento una fecha en formato de texto y la transforma a tipo Date para luego respetar el orden de 'dd/mm/aaaa'. Finalmente, la fecha será retornada en formato de texto
-
- Hacemos uso del hook **useEffect()** para hacer una única petición de la promoción vigente cuando la página se renderice, y a su vez seteamos la misma a una variable (**promotion**) del contexto Promotion para su acceso global. A su vez, hacemos una captura de los datos obtenidos en el caso de que todo salga bien y una captura del error en caso de que se presente algún inconveniente.

El componente App **retorna**:

```

68 return [
69   <div className="body">
70     <header className="header-container">
71       <div className="logo-container">
72         <img className="logo-image" src={logo} alt="logo_tap" width={150} />
73       </div>
74     </header>
75
76     <section className="desc-section-container">
77       {promotion != null
78       ?
79         promotion.expired === false
80         ?
81           promotion.couponsAvailables
82           ?
83             (<div className="desc-container-1 unselectable">
84               <div className="desc-container-2">
85                 <div className="desc-container-3">
86                   <div className="title-container">
87                     <h2 className="promotion-desc-title">{'${promotion.description} ${promotion.prizeMaxPrice}'}</h2>
88                   </div>
89                   <div className="desc-container">
90                     <h3 className="promotion-desc-subtitle">{'Tenés chance desde el ${formatDate(promotion.dateMin)} hasta el ${formatDate(promotion.dateMax)}'}</h3>
91                   </div>
92                 </div>
93               </div>
94             </div>)
95           :
96             (<div className="desc-container-1 unselectable">
97               <div className="desc-container-2">
98                 <div className="desc-container-3">
99                   <div className="title-container">
100                     <h2 className="promotion-desc-title">{'${promotion.noAvailableCouponsMessage}'}</h2>
101                   </div>
102                 </div>
103               </div>
104             </div>)
105           :
106             (<div className="desc-container-1 unselectable">
107               <div className="desc-container-2">
108                 <div className="desc-container-3">
109                   <div className="title-container">
110                     <h2 className="promotion-desc-title">{'${promotion.promotionExpiredMessage}'}</h2>
111                   </div>
112                 </div>
113               </div>
114             </div>)
115           : null
116       }

```

```

118   <div className="ranking-container">
119     <div className="ranking">
120       <Ranking />
121     </div>
122   </div>
123
124   {promotion != null
125   ?
126     <div className="promotional-image">
127       <img className="promotion-image" src={` ${getApiURL()} + promotion.Picture.formats.small.url`} alt="image_promo" />
128     </div>
129     : null
130   }
131 </section>
132 <section className="form-section-container unselectable">
133   {promotion && promotion.expired === false && promotion.couponsAvailables
134   ?
135     <a.div {...bind()} style={{ y, touchAction: 'none' }}>
136       <div className="form-swipeable-container">
137         <div className="form-div">
138           <div className="form-button-container" onClick={isClosed ? open : close}>
139             <div className="button-to-slide"></div>
140           </div>
141           <div className="form-container">
142             <div className="form-title-container">
143               <h2 className="form-title">{texts.FORM_TITLE}</h2>
144             </div>
145             <div className="formulary">
146               <Router>
147                 <Route path="/" exact>
148                   <Formulary />
149                 </Route>
150               </Router>
151             </div>
152           </div>
153         </div>
154       </div>
155     </a.div>
156     : null
157   }

```

```
158         {openCouponModal
159           ? <CouponMessage />
160           : null
161         }
162         {openTermsModal
163           ? <TermsMessage />
164           : null
165         }
166       </section>
167     </div>
168   );
169 }
170 }
```

El componente App retorna el esqueleto de la Landing Web, es decir:

- **Sección cabecera**

Basado en etiqueta semántica y div's, muestra el **logo de TAP**.

- **Sección descripción**

Basado en etiqueta semántica y div's, representa lo siguiente:

- Descripción de la **promoción**: se mostrará una descripción de la promoción en el caso de que la misma siga vigente. Si la promoción expiró, se mostrará un mensaje personalizado acorde a la situación descrita, al igual que si la cantidad de cupones se terminó también se mostrará un mensaje personalizado acorde a la situación descrita.
- **Ranking**: se usará el componente **Ranking** tratado anteriormente para que el Usuario pueda acceder al mismo mediante un botón.
- **Imagen**: se mostrará una imagen acorde a la promoción vigente si la misma existe.

- **Sección formulario**

Basado en etiqueta semántica y div's, muestra el formulario que el Usuario utilizará para el registro a la promoción vigente. Se mostrará sólo si la promoción sigue vigente y si hay cupones disponibles.

Como se aprecia, el componente deslizable está vinculado con la constante **'bind'** para el uso de las animaciones y gestos que personalizamos anteriormente.

Se usará el componente del formulario **'Formulary'** tratado anteriormente. El mismo se encuentra rodeado de otro componente **'Router'** debido a que es utilizado para capturar datos de la URL.

A su vez, haremos uso de los Componentes de ventanas Modal tipo pop-up: **'CouponMessage'** y **'TermsMessage'**.

Cada componente estará condicionado por un operador ternario debido a que solo deben ser mostrados cuando las variables booleanas globales de apertura de modales (**openCouponModal**, **openTermsModal**) estén en **'true'**.

Estos componentes no están dentro del componente Formulary o de algún div que rodee Formulary, debido a que si lo estuviesen se verían afectados por el div contenedor que vincula la constante **'bind'** para las animaciones.

Cabe destacar que cada etiqueta tratada en este componente App.js es identificada por un atributo de clase **'className'** para el uso de **estilos '.css'**.

```
172 export default function AppWrapper() {  
173   return (  
174     <PromotionState>  
175       <FormState>  
176         <App />  
177       </FormState>  
178     </PromotionState>  
179   )  
180 };  
181
```

Como se aprecia, hicimos uso de un **wrapper** para el componente App.js debido a que en el mismo debíamos hacer uso de las variables globales de los contextos definidos. Si el componente App.js no tenía como padres a los componentes **PromotionState** o **FormState** no tenía el acceso a sus propiedades.

Utils

A continuación mostraremos y explicaremos los componentes y archivos restantes de suma utilidad empleados en el proyecto:

- **Carpeta Context:**

El contexto de React proporciona datos a los componentes sin importar qué tan profundos estén en el árbol de componentes. El contexto se utiliza para gestionar datos globales, como por ejemplo: estado global, tema, servicios, configuración de usuario y más.

Usar el contexto en React requiere de 3 pasos:

1. Crear el contexto (createContext)
2. Proporcionar el contexto (.Provider)
3. Consumir el contexto (useContext)

A lo largo de esta carpeta veremos cómo haremos uso de este concepto de React aplicando todo lo necesario para poder implementarlo para su consumo.

Esta carpeta contendrá diversas carpetas referidas a los estados que queramos acceder a nivel global y un archivo types.js que contendrá los nombres de las funciones a ejecutar. Las carpetas internas siempre contendrán 3 archivos (referidos al Contexto, State y Reducer).

- **Carpeta Form:**

Está compuesta por 3 archivos que trabajan en conjunto: **FormContext**, **FormReducer** y **FormState**

- **FormContext**

```
1  import { createContext } from "react";
2
3  const FormContext = createContext();
4
5  export default FormContext
```


Importamos la función '**createContext**' de React la cual nos devuelve un objeto que almacenaremos en 'FormContext', el cual consta de dos partes:

- **Context.Provider:** todos los componentes que estén renderizados dentro de este padre podrán acceder al estado del contexto.
- **Context.Consumer:** serán los que consuman el estado del Provider, por lo general no hacemos uso de este sino directamente con el hook useContext que React nos proporciona.

Esta función es utilizada para que todo mi componente FormState sea un estado. Es una función que posibilita retornar un estado.

Utilizamos el FormContext para decir que todos los componentes que estén dentro de él van a poder acceder al estado definido en FormState.

- **FormReducer**

```
1 import { SET_FORM, SET_STEP, SET_REGISTERED_USER, SET_FORM_COMPLETED, SET_OPEN_COUPON_MODAL, SET_OPEN_TERMS_MODAL, SET_COUPON } from "../types";
2
3 export default function FormReducer(state, action) {
4   const { payload, type } = action
5
6   switch (type) {
7     case SET_FORM:
8       return {
9         ...state,
10        dataForm: { ...state.dataForm, ...payload }
11      }
12     case SET_STEP:
13       return {
14         ...state,
15        step: payload
16      }
17     case SET_REGISTERED_USER:
18       return {
19         ...state,
20        registeredUser: payload
21      }
22     case SET_FORM_COMPLETED:
23       return {
24         ...state,
25        formCompleted: payload
26      }
27     case SET_OPEN_COUPON_MODAL:
28       return {
29         ...state,
30        openCouponModal: payload
31      }
32     case SET_OPEN_TERMS_MODAL:
33       return {
34         ...state,
35        openTermsModal: payload
36      }
37     case SET_COUPON:
38       return {
39         ...state,
40        coupon: payload
41      }
42     default:
43       return state;
44   }
45 }
```

Importamos, de **types.js**, los nombres de las funciones que vamos a utilizar para actualizar el estado vigente.

Esta función que exportamos recibe por argumento el **estado** actual y un **'action'** que es un objeto el cual contiene: el **type** (nombre de la función que queremos ejecutar para actualizar el estado) y un **payload** (los datos externos que son recibidos).

Internamente se hará uso de un **switch** para decidir, dependiendo del **type** que se pase, que es lo que queremos hacer. Por ejemplo, en el caso de que el type sea 'SET_FORM' traeremos el estado vigente y a su vez actualizaremos la propiedad **dataForm** con los nuevos datos recibidos.

- **FormState**

```
1  import React, {useReducer} from 'react'
2  import FormContext from './FormContext';
3  import FormReducer from './FormReducer'
4
5  const FormState = (props) => {
6
7      const initialState = {
8          dataForm: {
9              name: '',
10             lastName: '',
11             email: '',
12             phone: '',
13             dni: '',
14             acceptTerms: false,
15         },
16         step: 0,
17         registeredUser: false,
18         formCompleted: false,
19         openCouponModal: false,
20         openTermsModal: false,
21         coupon: null
22     }
23
24     const [state, dispatch] = useReducer(FormReducer, initialState);
25
26     const setForm = (data) => {
27         dispatch({
28             type: 'SET_FORM',
29             payload: data
30         })
31     }
32
33     const setStep = (data) => {
34         dispatch({
35             type: 'SET_STEP',
36             payload: data
37         })
38     }
39
40     const setRegisteredUser = (data) => {
41         dispatch({
42             type: 'SET_REGISTERED_USER',
43             payload: data
44         })
45     }
46 }
```

```
47     const setFormCompleted = (data) => {  
48         dispatch({  
49             type: 'SET_FORM_COMPLETED',  
50             payload: data  
51         })  
52     }  
53  
54     const setOpenCouponModal = (data) => {  
55         dispatch({  
56             type: 'SET_OPEN_COUPON_MODAL',  
57             payload: data  
58         })  
59     }  
60  
61     const setOpenTermsModal = (data) => {  
62         dispatch({  
63             type: 'SET_OPEN_TERMS_MODAL',  
64             payload: data  
65         })  
66     }  
67  
68     const setCoupon = (data) => {  
69         dispatch({  
70             type: 'SET_COUPON',  
71             payload: data  
72         })  
73     }  
}
```

Importamos el **Contexto** y **Reducer** previamente explicado y el hook **useReducer** que es una alternativa a **useState**.

El **useReducer** nos permite definir qué función vamos a ejecutar (**Reducer**) y qué datos vamos a pasar (**initialState**). Al igual que **useState** tendremos un estado al cual podremos consumir y actualizar.

El **Reducer** vendría a ser las funciones que se van a ejecutar dependiendo de lo que yo le pase (**type y payload**), para eso haremos uso de la función '**dispatch**' (función que utilizamos para poder actualizar el estado).

El **Reducer** tiene distintas funciones posibles a ejecutar, por lo tanto, gracias al **dispatch** podemos decidir, mediante parámetro, que función queremos ejecutar para poder actualizar algo específico del estado.

Este archivo, **FormState**, contendrá:

- Un objeto **initialState**: este objeto contiene el estado que la Landing cargará en su inicio. Sus propiedades son las siguientes:
 - **dataForm**: objeto con todos los datos que el Usuario ingresará en el formulario más la aceptación de los términos y condiciones. Es usado como objeto inicial en el Formulario.
 - **step**: variable numérica que nos especifica el paso actual del formulario.
 - **registeredUser**: variable booleana que nos especifica si el Usuario se registró correctamente. Es usada para poder actualizar la lista de participantes en el Ranking una vez que un Usuario se registra a la campaña.
 - **formCompleted**: variable booleana que nos especifica si el Formulario fue completado. Es usada en los íconos de completado del paso a paso del Formulario.
 - **openCouponModal**: variable booleana que nos especifica si es momento de abrir la ventana modal de tipo pop-up. Es usada cuando un Usuario termina su registro en la campaña y se debe mostrar lo obtenido.

- **openTermsModal:** variable booleana que nos especifica si es momento de abrir la ventana modal de tipo pop-up. Es usada cuando un Usuario está en el segundo paso de su registro en la campaña y quiere leer los términos y condiciones.
- **coupon:** objeto que contiene los datos del cupón obtenido, es decir el cupón en sí y la URL de referido.

- **Uso del hook useReducer**

Para el manejo del estado que será accesible a todos los componentes hijos.

- **Funciones**

Estas funciones presentes, son las encargadas de actualizar el estado, junto al uso de **dispatch**, el cual nos permite decirle al **Reducer** qué función queremos ejecutar de acuerdo con el **type** establecido.

- Finalmente retornaremos:

```
75     return (  
76         <FormContext.Provider value={{  
77             dataForm: state.dataForm,  
78             step: state.step,  
79             registeredUser: state.registeredUser,  
80             formCompleted: state.formCompleted,  
81             openCouponModal: state.openCouponModal,  
82             openTermsModal: state.openTermsModal,  
83             coupon: state.coupon,  
84             setCoupon,  
85             setOpenCouponModal,  
86             setOpenTermsModal,  
87             setForm,  
88             setStep,  
89             setRegisteredUser,  
90             setFormCompleted  
91         }}>  
92         {props.children}  
93     </FormContext.Provider>  
94 )  
95 }  
96  
97 export default FormState
```

Retornaremos el Componente **FormContext.Provider** junto a la propiedad **value** que es la que indica a los componentes hijos el estado y las funciones a las cuales pueden acceder. Los componentes hijos en este proyecto, serán todos los componentes presentes.

- Carpeta Promotion:

Está compuesta por 3 archivos que trabajan en conjunto:

PromotionContext, PromotionReducer y PromotionState:

- **PromotionContext**

```
1  import { createContext } from "react";
2
3  const PromotionContext = createContext();
4
5  export default PromotionContext
```

Importamos la función '**createContext**' de React la cual nos devuelve un objeto que almacenaremos en '**PromotionContext**', el cual consta de dos partes:

- **Context.Provider:** todos los componentes que están renderizados dentro de este padre podrán acceder al estado del contexto.
- **Context.Consumer:** serán los que consuman el estado del Provider, por lo general no hacemos uso de este sino directamente con el hook `useContext` que React nos proporciona.

Esta función es utilizada para que todo mi componente **PromotionState** sea un estado. Es una función que posibilita retornar un estado.

Utilizamos el **PromotionContext** para decir que todos los componentes que estén dentro de él van a poder acceder al estado definido en **PromotionState**.

- **PromotionReducer**

```
1  import { SET_PROMOTION } from "../types";
2
3  export default function PromotionReducer(state, action) {
4      const { payload, type } = action
5
6      switch (type) {
7          case SET_PROMOTION:
8              return {
9                  promotion: payload
10             }
11          default:
12              return state;
13      }
14  }
```

Importamos, de **types.js**, el nombre de la función que vamos a utilizar para actualizar el estado vigente.

Esta función que exportamos recibe por argumento el **estado** actual y un **'action'** que es un objeto el cual contiene: el **type** (nombre de la función que queremos ejecutar para actualizar el estado) y un **payload** (los datos externos que son recibidos).

Internamente se hará uso de un **switch** para decidir, dependiendo del **type** que se pase, que es lo que queremos hacer. Por ejemplo, en el caso de que el type sea 'SET_PROMOTION' actualizaremos la propiedad promotion con los nuevos datos recibidos.

- **PromotionState**

```
1  import React, {useReducer} from 'react'
2  import PromotionContext from './PromotionContext.js';
3  import PromotionReducer from './PromotionReducer.js'
4
5  const PromotionState = (props) => {
6
7      const initialState = {
8          promotion: null
9      }
10
11     const [state, dispatch] = useReducer(PromotionReducer, initialState)
12
13     const setPromotion = (data) => {
14         dispatch({
15             type: 'SET_PROMOTION',
16             payload: data
17         })
18     }
```

Importamos el **Contexto** y **Reducer** previamente explicado y el hook **useReducer** que es una alternativa a **useState**.

El **useReducer** nos permite definir qué función vamos a ejecutar (**Reducer**) y qué datos vamos a pasar (**initialState**). Al igual que **useState** tendremos un estado al cual podremos consumir y actualizar.

El **Reducer** vendría a ser las funciones que se van a ejecutar dependiendo de lo que yo le pase (**type y payload**), para eso haremos uso de la función '**dispatch**' (función que utilizamos para poder actualizar el estado).

El **Reducer** tiene distintas funciones posibles a ejecutar, por lo tanto, gracias al **dispatch** podemos decidir, mediante parámetro, que función queremos ejecutar para poder actualizar algo específico del estado.

- Este archivo, **PromotionState**, contendrá:

- **Un objeto initialState**

Este objeto contiene el estado que la Landing cargará en su inicio. Su propiedad es la siguiente:

- **promotion**: objeto con todos los datos de la promoción vigente (descripciones, fecha mínima y máxima, máxima cantidad de participantes a visualizar en Ranking, premio mínimo y máximo, imagen de promoción, entre otros.). Es usado en los componentes **App.js**, **Ranking.js**, **CouponMessage.js** y **TermsMessage.js**.

- **Uso del hook useReducer**

Para el manejo del estado que será accesible a todos los componentes hijos.

- **Funciones**

Esta función presente, es, de momento, la única encargada de actualizar el estado vigente, junto al uso de **dispatch**, el cual nos permite decirle al **Reducer** qué función queremos ejecutar de acuerdo con el type establecido.

- Finalmente retornaremos:

```
20     return (  
21         <PromotionContext.Provider value={{  
22             promotion: state.promotion,  
23             setPromotion  
24         }}>  
25             {props.children}  
26         </PromotionContext.Provider>  
27     )  
28 }  
29  
30 export default PromotionState
```

Retornaremos el Componente **PromotionContext.Provider** junto a la propiedad value que es la que indica a los componentes hijos el estado y las funciones a las cuales pueden acceder. Los componentes hijos en este proyecto, serán todos los componentes presentes.

- **Archivo types.js**

```
1 export const SET_FORM = 'SET_FORM'
2 export const SET_STEP = 'SET_STEP'
3 export const SET_PROMOTION = 'SET_PROMOTION'
4 export const SET_REGISTERED_USER = 'SET_REGISTERED_USER'
5 export const SET_FORM_COMPLETED = 'SET_FORM_COMPLETED'
6 export const SET_OPEN_COUPON_MODAL = 'SET_OPEN_COUPON_MODAL'
7 export const SET_OPEN_TERMS_MODAL = 'SET_OPEN_TERMS_MODAL'
8 export const SET_COUPON = 'SET_COUPON'
```

El archivo **types.js** contiene todos los nombres exportados, en constantes, de las funciones a utilizar por los contextos presentes en el proyecto. Es decir, cada Estado con su correspondiente Reducer, hará uso de las funciones que lo integren y para poder invocarlas se las llamará con estos nombres asignados en constantes.

- **Carpeta Strapi:**

- **Archivo config.js:**

```
1  import { config } from "dotenv";
2
3  config()
4
5  function getAuth(params) {
6    if (params) {
7      return {
8        headers: {
9          Authorization: process.env.REACT_APP_AUTHORIZATION_STRAPI
10        },
11        params: params
12      }
13    } else {
14      return {
15        headers: {
16          Authorization: process.env.REACT_APP_AUTHORIZATION_STRAPI
17        }
18      }
19    }
20  }
21
22  function getApiURL() {
23    return process.env.REACT_APP_URL_STRAPI
24  }
25
26  function getPromotionId() {
27    return process.env.REACT_APP_PROMOTION_ID
28  }
29
30  export { getAuth, getApiURL, getPromotionId }
```

Cómo podemos apreciar, hacemos uso de la librería **'dotenv'** para poder cargar las variables de entorno presentes en el proyecto y hacer uso de estas en las siguientes funciones que luego serán exportadas para su uso:

- Función **getAuth()**: la misma recibe por parámetro un objeto que puede contener el email del Usuario que quiera consultar su posición y como requerido el ID de promoción, debido a que la consulta debe hacerse en la promoción correspondiente con sus participantes correspondientes. Finalmente, se retorna el objeto de autenticación solicitado.
- Función **getApiURL()**: esta función retorna la URL de Strapi almacenada en una variable de entorno.

- Función **getPromotionId()**: esta función retorna el ID de promoción vigente almacenado en una variable de entorno.

- Archivo **data.js**:

```
1 import axios from 'axios'
2 import { getAuth, getApiURL, getPromotionId } from './config.js';
3
4 async function getPromotion() {
5   const { data, status } = await axios.get(`${getApiURL()}/promotions/${getPromotionId()}`, getAuth());
6   if (status !== 200) {
7     throw new Error(data.message)
8   }
9   return data
10 }
11
12 async function getAllParticipants({ promotionId, email }) {
13   const { data, status } = await axios.get(`${getApiURL()}/ranking`, getAuth({ promotionId, email }));
14   if (status !== 200) {
15     throw new Error(data.message)
16   }
17   return data
18 }
19
20 async function getCoupon({ referr, req }) {
21   const { data } = await axios.post(`${getApiURL()}/participants?referr=${referr}`, {...req, promotionId: getPromotionId()}, getAuth())
22   if (data.status !== 201) {
23     throw new Error(data.message)
24   }
25   return data
26 }
27
28 export { getPromotion, getAllParticipants, getCoupon }
```

Cómo podemos apreciar, hacemos uso de la librería **'axios'** para hacer las peticiones necesarias y además de las funciones que nos provee el archivo **config.js** (explicadas anteriormente). Las funciones exportadas que contiene este archivo nos proveen todas las variables de negocio utilizadas en el proyecto (**promoción, participantes y cupones**):

- **getPromotion()**: hacemos una petición con axios para poder obtener todos los datos referidos a la promoción vigente, el mismo nos traerá el registro correspondiente a la promoción actual con todas sus propiedades. En el caso

de que el status de lo obtenido con axios sea distinto del código de estado 200 se generará el correspondiente error.

- **getAllParticipants():** esta función recibe como argumento un objeto con la propiedad requerida '**promotionId**' y como propiedad opcional 'email'. Internamente hará una petición con axios para obtener a todos los participantes de la promoción vigente. En el caso de que el status de lo obtenido con axios sea distinto del código de estado 200 se generará el correspondiente error.
- **getCoupon():** esta función recibe como argumento un objeto con las propiedades '**referr**' (referido) y '**req**' (inputs del usuario en el formulario). Internamente hará un post con axios en el cual se le pasa la URL de Strapi junto a la ruta de 'participants' más el parámetro de consulta del referido; el objeto con los datos ingresados por el Usuario en el formulario, el ID de la promoción vigente y el objeto de autenticación solicitado por Strapi . En el caso de que el status de lo obtenido con axios sea distinto del código de estado 201 se generará el correspondiente error.

- Carpeta Assets:

- Archivo strings.js


```
1  export const texts = {
2    COUPON_MC_BUTTON: 'Copiar cupon',
3    LINK_MC_BUTTON: 'Copiar link',
4    TEXT_COPIED: 'Copiado.',
5    FORM_TITLE: '¡Ingresa tus datos para participar!',
6    FORM_LABEL_NAME: 'Ingresa tu nombre',
7    FORM_LABEL_LASTNAME: 'Ingresa tu apellido',
8    FORM_LABEL_DNI: 'Ingresa tu dni',
9    FORM_LABEL_EMAIL: 'Ingresa tu email',
10   FORM_LABEL_PHONE: 'Ingresa tu telefono',
11   FORM_LABEL_TERMS: 'He leído y acepto los ',
12   MAX_CHARACTER_TEXT_VAL: 'No se permiten más de 20 caracteres.',
13   ONLY_ALPHABET_TEXT_VAL_NAME: 'Sólo se permite texto de 2 a 20 caracteres.',
14   ONLY_ALPHABET_TEXT_VAL_LASTNAME: 'Sólo se permite texto de 2 a 20 caracteres y un apostrofe en segunda posicion.',
15   REQUIRED_TEXT_VAL: 'No es posible dejar el campo vacío.',
16   MIN_NUMBER_DNI_VAL: 'Su DNI debe ser menor a 100.000.000',
17   MAX_NUMBER_DNI_VAL: 'Su DNI debe ser mayor a 10.000.000',
18   DNI_VAL: 'El DNI ingresado no es válido.',
19   EMAIL_VAL: 'El email ingresado no es válido.',
20   PHONE_VAL: 'El número ingresado no es válido.',
21   NEXT_BUTTON: 'Siguiente',
22   BACK_BUTTON: 'Volver',
23   SEE_POSITION_BUTTON: 'Ver mi posición',
24   TITLE_OF_TERMS: 'Términos y Condiciones',
25   PRIZES_BUTTON: 'Ver premios',
26   TITLE_OF_PRIZES: 'Premios',
27   RANKING_BUTTON: 'Ver ranking',
28   TITLE_OF_RANKING: 'Ranking',
29   SUBTITLE_OF_RANKING: '¡Ingresa tu correo electrónico para saber tu posición!',
30   DESCRIPTION_OF_THE_RANKING_POSITION: 'Estás en la posición',
31   DESCRIPTION_OF_THE_PRIZE_IN_RANKING: 'Tu premio es de',
32   FORM_REGISTRATION_ERROR: 'Error, no se pudo realizar el registro en la campaña.',
33   ERROR_NO_CURRENT_PROMOTION: 'No hay promoción vigente',
34   ERROR_NO_REGISTERED_USERS: 'No hay usuarios registrados',
35   NO_PARTICIPANTS: 'Aún no hay participantes ¡Sé el primero en sumarte!',
36   RANKING_WITHOUT_COUPONS: 'Aún no hay participantes.',
37   EMAIL_NOT_FOUND: 'Email no encontrado',
38   INVITATION_TO_PARTICIPATE: '¡Te invitamos a completar el formulario para participar!'
39 }
```

El mismo contiene todas las cadenas de caracteres que serán usadas a lo largo de todo el proyecto.