Calle Solidaria:
Ayudar y recibir ayuda nunca fue más fácil

Abstracto. Los centros de ayuda comunitarios siempre han formado parte de nuestras ciudades, sin embargo, frente al panorama económico estos centros han tomado cada vez más protagonismo. Existen dos factores cruciales de los que necesita un centro de ayuda para poder funcionar correctamente: la difusión de sus iniciativas entre la comunidad y la invaluable ayuda de voluntarios para sostener estos esfuerzos. En el presente informe, planteamos como fue el proceso para crear Calle Solidaria, una plataforma web desarrollada utilizando distintas tecnologías, APIs y metodologías de desarrollo que busca atacar estas dos problemáticas que todo centro de ayuda debe afrontar para poder ayudar a la mayor cantidad de personas.

Abstract. Community assistance centers have long been an important part within our urban landscapes; however, in the face of the prevailing economic climate, these centers have assumed an increasingly prominent role. There are two critical factors upon which an assistance center must rely in order to function effectively: the dissemination of its initiatives within the community and the invaluable support of volunteers to sustain these endeavors. In the present report, we shall outline the process of creating Calle Solidaria, a web-based platform developed through the utilization of diverse technologies, APIs, and development methodologies, which seeks to address these two challenges that every assistance center must confront in order to provide aid to the greatest number of individuals.

Keywords: Solidarity, Charity, Refugees, Shelters, Volunteering, NGOs, Web development, Flask, Python, Javascript, Mapbox, HTML, CSS, Bash, SQL, Agile.
Solidaridad, Caridad, Refugiados, Refugios, Voluntariado, ONGs, Desarrollo web,Flask, Python, Javascript, Mapbox, HTML, CSS, Bash, SQL, Metodologías ágiles.

Introducción

Hoy en día frente a la delicada situación en la que se encuentra nuestro país donde el porcentaje de pobreza supera el 50% de la población y que cuando se hace foco en los niños y adolescentes esta cifra asciende al 70% [1], los centros de ayuda comunitarios han tomado particular relevancia para el sostén del tejido social. Sin embargo, si bien existe una gran cantidad de refugios y comedores comunitarios mantenidos en gran parte por voluntarios que desinteresadamente ofrecen de su tiempo y servicios para ayudar a los que menos tienen, consideramos que muchas veces estas iniciativas no tienen la difusión o alcance suficiente que la coyuntura amerita.

Es por esto que el grupo 404 se enfocó en desarrollar una solución que busque reducir la brecha entre gente dispuesta a ayudar al prójimo y gente que necesita de esta ayuda. En este documento podrá observar a detalle el recorrido que tomó el grupo para crear Calle Solidaria, un proyecto enfocado en, por un lado, darle visibilidad a aquellos refugios que se encuentran en las distintas ciudades del mundo para que aquellos en situaciones de necesidad puedan conocerlos y acceder a información relevante de ellos de forma rápida y completamente gratuita, y por otro lado, funcionar como una plataforma que conecte refugios y ONGs que necesiten de ayuda con personas que estén dispuestas a colaborar siendo voluntarios.

Comentaremos en distintas secciones cuales fueron nuestras motivaciones iniciales, cuales ideas habíamos planteado en un inicio y como fuimos moldeando Calle Solidaria hasta la versión que el presente documento detalla. Comentaremos cuáles fueron las tecnologías implementadas, que herramientas y APIs externas utilizamos y qué metodologías de trabajo utilizamos durante el proceso.

Solución Propuesta

Nuestra solución final no fue considerablemente distinta a la que habíamos imaginado en un inicio. Para crear la página Calle Solidaria, debíamos crear una plataforma que permite a refugios en cualquier parte del planeta registrase en nuestra página, la cual les permite a dichos refugios agregar información relevante sobre ellos, su misión, su ubicación y demás detalles relevantes con el principal objetivo de tener mayor visibilidad para que aquellos que los necesiten los puedan conocer a estos refugios mediante las vistas de feed o de mapa que provee Calle Solidaria, o que quienes quieran se

puedan sumar a las distintas causas puedan hacerlo a través de la funcionalidad de voluntariado que permite a una persona inscribirse al refugio en el cual quiere colaborar.

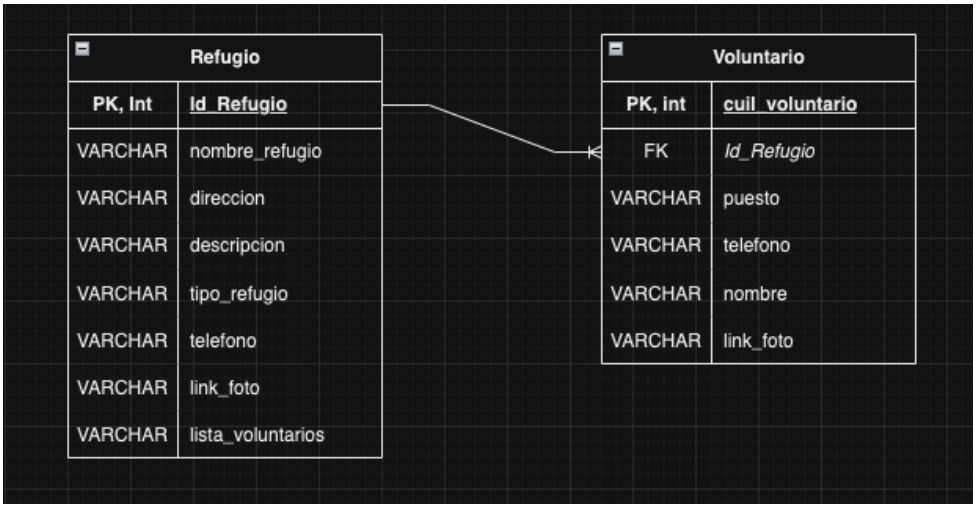

Figura 1. Visualización de bases de datos presentes en Calle Solidaria. Un refugio puede tener muchos voluntarios, los cuales se encuentran listados. Un voluntario no se puede inscribir en más de un refugio.

En tanto al código y las aplicaciones que componen a Calle Solidaria, la misma consiste de una página web que actúa como el front end de la aplicación y una API que se comunica con el front y que también gestiona la información de los refugios y voluntarios mediante una base de datos. El front end de la página consiste en distintas vistas HTML con distintos estilos CSS que funcionan en conjunto mediante Flask. Algunas de estas vistas tienen cierta lógica desarrollada en Javascript pero principalmente sirven como intermediarias entre el usuario y nuestra API de Calle Solidaria.

La API o el backend de Calle solidaria consiste fundamentalmente de una aplicación Flask y una base de datos SQL que mediante distintos métodos como GET, POST o DELETE nos permite interactuar con la información que tenemos almacenada en la base de datos de refugios y de voluntarios, para así poder guardar nuevos refugios, eliminarlos, etc.

Si bien la estructura de Calle Solidaria es relativamente sencilla, a lo largo del proyecto y como en cualquier proyecto nuevo se nos presentaron ciertas dificultades durante el desarrollo, no solo a nivel código sino que también a nivel organizativo. Al principio, subestimamos nuestras capacidades y nuestro alcance fijado hacia el primer entregable fue muy básico ya que principalmente consistio de vistas estáticas HTML donde habían pocas tareas para repartir entre muchos miembros y que al ser terminadas nos dejaba con un proyecto que si bien a simple vista se veía aceptable, por detrás no contaba con la gran mayoría de funcionalidades que habíamos pensado para Calle Solidaria. Sin embargo, conforme pasaron los días proseguimos con la implementación de la API y poco a poco logramos darle la funcionalidad que habíamos querido desde un primer momento.

Como anteriormente mencionamos, durante todo el proceso surgieron algunos desafíos o problemas que tuvimos que afrontar para completar la plataforma. El primer problema y más

importante, ya que forma parte del núcleo de la página, fue como íbamos a hacer la vista del mapa, no por su implementación en las vistas, sino porque cumpla su objetivo de mostrar los refugios existentes en el mismo. Tras algunos días de investigar y probar distintas alternativas, librerías y APIs externas, decidimos por utilizar Mapbox GL JS [2], una API que nos permite con muy pocas líneas de Javascript, crear insertar un mapa a nuestra web, agregar controles y, lo más importante, agregarle distintos puntos en el mapa (lo que en Mapbox se conoce como circles), que al apoyar el mouse sobre ellos muestre mediante un popup el nombre y la dirección del refugio que el usuario estaba viendo.

```
// Creo un nuevo mapa centrado en Buenos Aires.
const map = new mapboxgl.Map({
    container: 'map',
    style: 'mapbox://styles/mapbox/streets-v12',
    center:[-58.4339192, -34.6020498],
    zoom: 11
});

// Agrego menu de navegacion al mapa.
const nav = new mapboxgl.NavigationControl()
map.addControl(nav)

map.on('load', () => {
    map.resize()
    // Actualizo archivo geojson para que tenga descripciones
    var geojsonConDescripciones = generateDescriptions(geojson_refugios);

    // Agrego a mapa el archivo geojson.
    map.addSource('refugios', {
        'type': 'geojson',
        'data': geojsonConDescripciones
    });

    // Agrego una capa al mapa que muestre los lugares del geojson
    map.addLayer({
        'id': 'refugios',
        'type': 'circle',
        'source': 'refugios',
        'paint': {
            'circle-color': '#4264fb',
            'circle-radius': 6,
            'circle-stroke-width': 2,
            'circle-stroke-color': '#ffffff'
        }
    });
});
```

Figura 2. Código Javascript que muestra cómo se crea un mapa en Mapbox y como se agregan refugios mediante un geojson.

Para poder agregar los distintos refugios al mapa en forma de Circles, tras crear el mapa base, le agregamos una layer o capa en Mapbox la cual contendrá todos los circles o puntos que representan nuestros refugios. Para poder agregar estos circles en las coordenadas necesarias y poder mostrar el nombre y dirección de los refugios, procesamos un archivo GeoJson que contendrá las coordenadas específicas donde se ubica el refugio, y ciertas propiedades, entre las cuales se encuentran el nombre y la dirección de dicho refugio.

```
"features": [
    {
        "type": "Feature",
        "properties": {
            "description": "comedor",
            "marker-symbol": "",
            "title": "Comedor del Movimiento Popular",
            "address": "Bonpland 1660, Palermo, Buenos Aires, 1440, Argentina"
        },
        "geometry": {
            "type": "Point",
            "coordinates": [
                -58.438025964507545,
                -34.58350515366307
            ]
        }
    },
```


Figura 3. Ejemplo de GeoJson con información de un comedor.

Tras haber probado cómo funcionaban las capas de Mapbox y cómo interactuar con los GeoJsons, procedimos a crear una funcionalidad para crear estos GeoJsons cada vez que se cargaba la vista del mapa mediante una request a la API que traía los datos de todos los refugios almacenados en la base de datos, les daba el formato GeoJson que Mapbox necesita y finalmente los agrega al mapa como anteriormente mencionamos en forma de una nueva capa. Algo que también vale la pena mencionar es que para obtener las coordenadas en las que se debería ubicar un circle en el mapa, utilizamos la API de Mapbox llamada Geocoding [3] que nos permite obtener coordenadas geográficas a partir de la dirección postal que nos proveyó el usuario al registrar el refugio. Además, utilizamos la API de Mapbox Directions [4] para poder obtener direcciones desde cualquier punto del mundo hacia un refugio en particular. Si bien no creemos que sea óptimo pedir todos los refugios a la base de datos cada vez que se carga la vista del mapa y se podría hacer optimizaciones para cachear el último resultado y solo pedir cualquier nuevo refugio que se haya agregado, para los fines sencillos de Calle Solidaria decidimos no sobre optimizar esta cuestión para enfocarnos en otros inconvenientes que fueron surgiendo durante el desarrollo del proyecto.

El segundo problema, que consideramos la otra parte del núcleo de nuestro proyecto, era la inscripción de un voluntario en el refugio, como hacer que una vez que se cree el voluntario, este aparezca en la lista de su respectivo refugio. Como la lista es de tipo Varchar, en principio no había una visión clara de cómo manejar el campo como una lista de verdad, hasta que luego de un periodo de investigación, se llegó a una solución, la cual era convertir el Varchar en una lista de Python, manejarla como lista y una vez hechas las operaciones volverla a convertir en su tipo original. Esto se pudo realizar utilizando los métodos json.loads() para convertir “[]” => [] y json.dumps() para convertir [] => “[]”

```
seleccionar_refugio = text("""SELECT * FROM refugios WHERE nombre_refugio = :nombre_refugio;""")
```

Figura 4. Query para obtener un refugio. (En el formulario el voluntario ingresa el nombre del refugio)

```
update_refugio = text(""" UPDATE refugios SET lista_voluntarios = :lista_voluntarios
            WHERE id_refugio = :id_refugio;
""") #Updatea la LISTA_VOLUNTARIOS de la lista de voluntarios
```

Figura 5. Query para actualizar la lista del refugio.

```
try:
    refugio = conn.execute(seleccionar_refugio,{
        'nombre_refugio': volunteer["nombre_refugio"]
    }).fetchone() #Obtiene un refugio por nombre

    if not refugio:
        return jsonify({'message': 'No existe un refugio con ese nombre'}), 404
```

Figura 6. Ejecución del query para buscar refugio manejando el error por si no encuentra uno que exista con tal nombre

```
if refugio[7] == None: #Si no habia un voluntario antes se crea una lista
    lista_voluntarios = [volunteer["cuil_voluntario"]]
else:
    lista_voluntarios = json.loads(refugio[LISTA_VOLUNTARIOS]) #Convierte "[]" => []
    lista_voluntarios.append(volunteer["cuil_voluntario"])

lista_voluntarios = json.dumps(lista_voluntarios) #Convierte [] => "[]"


conn.execute(update_refugio,{'id_refugio': refugio[ID_REFUGIO],'lista_voluntarios': lista_voluntarios})
```

Figura 7. Si encuentra un refugio y su campo 7 que es la lista_voluntarios está vacía, crea una lista con el cuil del nuevo voluntario. En caso de ya existir agarra la lista_voluntarios de tipo Varchar, la convierte en una lista en python para appendear el nuevo voluntario, la vuelve a convertir en su tipo original, y ejecuta la query para actualizar con la nueva lista.

A la hora de conectar el front-end con la API, en concreto cómo conectar los formularios de registro tanto de voluntario como de refugios, por un momento nos quedamos sin saber muy bien que hacer, pero luego de una ardua búsqueda y basándonos en lo que nuestro docente nos propuso como consigna averiguamos cómo hacer para que, a través de un archivo formato JSON, se envíe la información que el front recibía en los formularios de por ejemplo registro de refugios o de registro de voluntarios, para que la misma viaje hacia la API y esta información se guarde en la base de datos o se modifique de acuerdo a la necesidad específica del método en cuestión que se quiera implementar. Este tipo de endpoints contaban con distintos paths y verbs como GET, POST o PATCH.

En el siguiente fragmento de código se puede ver como implementamos esto para la funcionalidad de edición de refugio, la cual también consiste de un formulario en el que el usuario ingresa la información que desea modificar, a esta información se le da formato JSON y se envía a la API para ser procesada.

```python
@app.route("/edicion_refugio/<id>",methods = ["GET","POST"])
def edicion_refugio(id):
    if request.method == "POST":
        nombre_refugio, direccion, descripcion, tipo, telefono, usuario, foto = data_return("refugio")
        datos = {
            "nombre_refugio": nombre_refugio,
            "direccion": direccion,
            "descripcion": descripcion,
            "tipo_refugio": tipo,
            "telefono": telefono,
            "link_foto": foto
        }
        datos_json = json.dumps(datos)
        URL = "http://pedrogillen.pythonanywhere.com/refugios/"+id
        res = requests.patch(URL, data=datos_json, headers={'Content-Type': 'application/json'})
        if res.status_code == 200:
            return redirect(url_for('detalles_refugio', id=id))
    return render_template("editar_refugio.html", id=id)
```

Figura 8. Funcion de app.py en el front end que muestra la vista edicion_refugio y procesa la información recibida.

Otro problema más en el que debatimos bastante pero que finalmente decidimos no implementar, fue la autenticación de "Usuarios" a través o un sistema de registro con contraseñas o a través de un token, que usariamos para poder modificar y eliminar tanto refugios como voluntarios. Las funcionalidades están hechas, pero dado el tiempo que nos consumía y la relevancia que se le daría para la entrega del TP, optamos por dejarlo para desarrollar a futuro y fuera de la materia, dejándonos un índice de hacia donde vamos a continuar.

Pruebas y/o validación

- Validación de entradas:
    - Tanto en refugios como voluntarios, todos los datos a excepción de las imágenes son campos obligatorios. Los usuarios tienen imagen por default y en el caso de los refugios, su lista se crea automáticamente vacía.
    - Los voluntarios no se pueden inscribir más de una vez, teniendo como clave primaria su CUIL.
- Verificación de funcionamiento:
    - Se puso en prueba cada servicio ofrecido por la API a través de Postman y está verificado su funcionamiento.
    - Se realizaron pruebas manuales a la vista del mapa para cada punto presente en él, verificando que la información se mostraba correctamente.
- Integración de todos los componentes:

    - Luego de la verificación de funcionamiento exclusivamente desde el back-end, se puso en prueba la llamada hacia los servicios desde el front-end, la cual salió a la perfección.
- Pruebas de Carga de datos:
    - Se probaron múltiples posibilidades de ingreso de datos, con diferentes características y se arreglaron todos los problemas relacionados con las mismas.
- Manejo de errores:
    - Simulamos posibles errores que se puede encontrar un usuario como entradas incorrectas, para confirmar que el sistema maneje las situaciones de manera adecuada utilizando también un mensaje claro hacia el usuario sobre el error

Plan de actividades

El desarrollo de este proyecto fue dividido en dos repositorios en la plataforma de GitHub donde uno corresponde al apartado del Front-End y el otro la Api y el Backend. Teniendo eso en cuenta, las tareas y actividades correspondientes fueron acordes a un repositorio u otro dependiendo de su funcionalidad. Las actividades están planteadas en la plataforma de Trello con su ID correspondiente y con una asignación a uno o varios miembros del equipo. Las estimaciones, por su parte, a cada una de las tareas fueron llevadas a cabo según el equipo completo teniendo en cuenta no sólo la dificultad de la tarea sino también el tiempo que pudiera demorar en un principio.

Respecto al desarrollo del trabajo previo al primer entregable, nuestra lista de funcionalidades en el Product Backlog se centró principalmente en la funcionalidad y el diseño del front-end de la página. Antes de elaborar tareas por primera vez para volcar en nuestro primer backlog, durante la reunión inicial de Sprint Planning, discutimos de manera general el propósito de nuestra página web, cómo interactuarían los usuarios con ella, y qué datos serían necesarios para nuestra base de datos.

Unos días después de haber definido nuestros alcances, en nuestra segunda reunión, realizamos una sesión de Planning Poker para obtener una estimación precisa y consensuada del esfuerzo requerido para completar las tareas. Utilizando Trello, definimos y asignamos las tareas, otorgándoles un puntaje según su grado de complejidad y esfuerzo.

Primero se planteó la estructura del proyecto en el Front con las carpetas templates, static y el archivo app.py acompañado de un script en Bash init.sh para la instalación de las dependencias y módulos necesarios para la implementación de la aplicación en Flask. Con el paso de los días, se fueron creando las distintas vistas dentro de la carpeta templates, se fue profundizando el archivo app.py para

el manejo de vistas, de los respectivos formularios y de las vistas de los errores y también se implementó el script sql para la Base de Datos.

Tras la elección de un template base para nuestro proyecto se fueron armando con el paso de los días las siguientes vistas:

En primer lugar, se armaron los documentos HTML utilizando formularios [5] como por ejemplo (para el cargado de refugio, registro de voluntario, editar refugio y editar voluntario).

Posteriormente, se prepararon los templates para el feed (mostrar todos los refugios), asimismo como las vistas de “detalles refugio” y “detalles voluntario” (que muestra específicamente todos los datos que se cargaron correspondientes a un refugio).

Y ya por último se estructuró la vista del mapa asociada a la api de mapbox con su archivo JavaScript para su personalización y también las vistas de los errores.

Por lo tanto, para nuestro segundo entregable nos centramos en el armado de la estructura completa del front-end con todas sus vistas y con la navegación entre ellas desde una barra de navegación.

Posteriormente, luego del segundo entregable hicimos foco principal en la página del lado del back-end, trabajando desde la api. Volvimos a hacer un sprint planning, donde definimos y asignamos las tareas con puntaje para esta parte del trabajo. Una vez asignadas, fuimos volcando nuestras tareas en la rama “develop” del repositorio, para ir complementando las tareas mutuamente e ir agregando la funcionalidad al back-end. Durante este sprint también tuvimos reuniones y discusiones donde definimos la implementación de la base de datos mediante MySQL. En primer lugar usamos la herramienta xampp, pero posteriormente en siguientes reuniones nos terminamos decantando por utilizar pythonanywhere. Nuestra siguiente reunión luego de haber finalizado nuestras tareas fue verificar el correcto funcionamiento de la api con la base de datos. Para ello de manera grupal utilizamos algunas herramientas como postman para ver la correcta implementación de todos los métodos en la base de datos [6].

Hablando específicamente del apartado de la API, comenzamos conectando a la misma la base de datos para poder llevar a cabo las implementaciones de todas las funciones. En primer lugar, se planteó la función, que utilizando el método get y haciendo una consulta a la base, devuelve un JSON en formato GEOJSON que es necesario para la utilización del mapa y agregado de puntos. Este objeto está estructurado de tal manera que se plantea una Feature Collection con los distintos refugios estableciéndose como puntos usando las coordenadas. Luego se llevaron a cabo las funciones que están relacionadas a cada vista HTML implementadas en el segundo entregable en el FRONT.

Uno de los principales obstáculos que se nos presentaron fue la forma de conectar la api del back-end con nuestro trabajo del front-end. Para ello requerimos de bastante indagación y trabajo en conjunto para poder luego asignarnos estas nuevas tareas y poder implementarlas. Una vez hecho esto, en nuestra próxima reunión verificamos el correcto funcionamiento de la página web y corregimos los mínimos detalles.

Documentación técnica del sistema:

Back-end: Endpoints.

- **‘obtener_refugios_geojson’: GET**
    1. Selecciona todos los registros en la tabla de refugios de la base de datos:
    2. Por cada refugio se hace un solicitud a la API mapbox (que dada una dirección detallada devuelve sus coordenadas), se chequea que la API haya encontrado las coordenadas para la dirección especificada.
       Se crea un diccionario con la descripción del refugio, incluidas las coordenadas y este se agrega a una lista que contiene objetos del mismo tipo.
    3. Si no hubo ningún error mientras se ejecutaba la petición se devuelve la lista con la información y coordenadas de cada refugio.
- **‘obtener_refugios’: GET**
    1. Selecciona todos los registros en la tabla refugios de la base de datos.
    2. Por cada refugio devuelto se crea un diccionario con sus propiedades (una por columna) y se lo agrega a una lista.
    3. Si no hubo ningún error durante la ejecución de la petición se devuelve la lista con los diccionarios que contienen la información de cada refugio.
- **‘obtener_refugio/<id>’: GET**
    1. Este endpoint es dinámico. El id no siempre es el mismo y depende de qué refugio se requiera. El id se va a usar para distinguir a un refugio específico de otros, teniendo en cuenta que los ids son únicos en la base de datos.
    2. Selecciona el refugio de la base de datos que coincide con el id especificado. En caso de no existir tal refugio se devuelve un error 404 indicando que no se encontró el recurso que se buscaba.

3. Crea una lista con los CUILs de los voluntarios de ese refugio (en caso de no haber voluntarios la lista se crea vacía)
4. Selecciona todos los voluntarios que coincidan con los CUILs de la lista y crea una nueva lista de listas en la que cada elemento es la información de un voluntario.
5. Si hubo un error se devuelve un error 500, si no se devuelve un objeto JSON con dos claves: 'data-refugio' que es un diccionario que contiene toda la info del refugio exceptuando los voluntarios y 'voluntarios' que es la lista de listas mencionada en el punto 4.

- **'crear_refugio': POST**
    1. Recibe en el body de la petición un objeto JSON con la información necesaria del refugio.
    2. Se inserta un nuevo refugio en la tabla refugios con la información recibida.
    3. En caso de haber un error de ejecución se devuelve un código 500, si no se devuelve un 201 indicando que se creó un recurso.
- **'crear_voluntario/': POST**
    1. Se selecciona el refugio donde el voluntario desea inscribirse, si no existe se devuelve 404
    2. Se inserta un nuevo voluntario en la tabla de voluntarios con la información obtenida desde el body de la petición y además el id del refugio al que se lo inscribe.
    3. Usando los métodos dumps y loads del módulo json se añade el CUIL del voluntario a la lista de voluntarios del refugio. En caso de no haber voluntarios en ese refugio se crea dicha lista.
    4. Si hubo un error en los pasos anteriores se devuelve 500, si no 201 indicando que se crearon recursos.
- **'obtener_voluntario/<cuil>': GET**
    1. Este endpoint es dinámico. El CUIL no siempre es el mismo y depende de qué refugio se requiera. El CUIL se va a usar para distinguir a un voluntario específico de otros, teniendo en cuenta que los CUILs son únicos en la base de datos.
    2. Selecciona el voluntario que tiene el CUIL especificado si no existe devuelve 404.
    3. Con la información del voluntario disponible se selecciona el refugio al que está inscripto el voluntario.
    4. En caso de no haber habido errores se devuelve toda la info del voluntario solo que se reemplaza el apartado de id_refugio por nombre_refugio.
- **'eliminar_refugio/<id>': DELETE** (Este endpoint no es accesible desde el front.)
    1. Selecciona el refugio que tiene de id el id especificado. En caso de no existir devuelve 404. (Esto se hace para verificar que efiectivamente se está eliminando un refugio).
    2. Se eliminan voluntarios del refugio.

3. Se elimina el refugio.

- **‘eliminar_voluntario/<cuil>: DELETE** (Este endpoint no es accesible desde el front.)
    1. Selecciona el voluntario con el cuil especificado, si no existe devuelve 404.
    2. Con la información del voluntario seleccionado, más específicamente con el id del refugio al que está inscripto el voluntario, se selecciona el refugio.
    3. Del refugio seleccionado se elimina el CUIL del voluntario de la lista de voluntarios.
    4. Se hace un update del refugio con la nueva lista de voluntarios.
    5. Se elimina el voluntario.
- **‘refugios/<id>’: PATCH**
    1. Obtiene desde el body los nuevos datos del refugio que se desea modificar.
    2. Selecciona el refugio que tiene como id el id especificado en el endpoint dinámico, si no existe devuelve 404.
    3. Hace un update del refugio con los nuevos datos.
- **‘/modificar_voluntario/<cuil>’: PATCH**
    1. Se obtiene el cuil mediante la URL de la request enviada a la api y los datos a modificar dentro del body de la query.
    2. Con el nombre del refugio se obtiene el id que le corresponde al refugio y devuelve un error 404 si no se encuentra ningún refugio.
    3. Se valida que el cuil ingresado exista y en caso de que no, devuelve un error 404.
    4. Hace un update de los datos con los recibidos en el body.

Front-end: Vistas.

- **Vistas de errores:** (Todas contienen un texto simple indicando el tipo de error)
    1. 400
    2. 403
    3. 404
    4. 500
- **“/”:**
    1. Hace una request a la base de datos al endpoint “/obtener_refugios” para obtener todos los datos de todos los refugios y se utiliza la función loads de json para pasarlos a un formato de lista de diccionarios.
    2. Dependiendo de si el método fue “GET” o “POST” se le pasa a la vista “feed.html” una lista de diccionarios. Si el método fue “GET”, se le pasa la lista sin filtrar, y si el método fue “POST”, se le pasa la lista filtrada por el texto recibido en la barra de búsqueda.

3. La vista “feed.html” muestra por cada refugio recibido una carta con toda su información y cada carta es un link hacia la vista “detalles_refugio.html” con la información del refugio correspondiente.

- **“/detalles_refugio/<id>”:**
    1. Mediante la url recibe el id del refugio a buscar y hace una request al endpoint “/obtener_refugio/<id>” de la api.
    2. Con los datos recibidos (info del refugio e info de los voluntarios), se los pasa a la vista “detalles_refugio.html”, la cual muestra todos los datos recibidos.
    3. La vista contiene un botón que te lleva hacia el mapa para saber cómo llegar. Otro para redirigir hacia la edición de refugios, y por cada voluntario tiene un link que te lleva hacia la vista con la información respectiva de cada voluntario.
- **“detalles_voluntario/<cuil>”**
    1. Se recibe el cuil del voluntario a buscar, y realiza una resquest de tipo get al endpoint de la API especificado por el URL (“obtener_voluntario”/<id>.
    2. La información recibida la transforma en un objeto Python mediante la función json.loads(), y luego renderiza a la página “detalles_voluntario.html”, pasándole además como parámetro el cuil del voluntario y la información correspondiente.
    3. La vista muestra toda la información que recibió por parámetro, y además contiene un botón que permite editar dicha información.
- “**cargar_refugio”**
    1. En este caso, la función cargar_refugio acepta dos métodos: GET y POST.
    2. Si recibe un POST, crea un diccionario llamado “datos”, en donde se guarda toda la información cargada por el usuario. Luego, ese diccionario es transformado a un objeto JSON.
    3. Se envía el diccionario en formato JSON en el body de la request al endpoint “crear_refugio”, y luego, se redirige al feed en donde se muestra la información del nuevo refugio.
    4. Si recibe un GET, lo que hace es renderizar la vista “cargar_refugio.html”.
- “**cargar_voluntario”**
    1. Al igual que cargar_refugio, la función cargar_voluntario puede recibir los métodos GET y POST.
    2. Si el método es POST, guarda toda la información cargada por el usuario en un diccionario, y convierte ese diccionario en un objeto JSON, para mandarlo por el body de la request.
    3. Luego de haber realizado todo esto, la función redirige al feed, donde se mostrará el refugio con el nuevo voluntario.

4. Si el método es GET, renderiza a la vista “cargar_voluntario.html”, que es un formulario para cargar los datos.

- “**mapa”**
    1. La función mapa renderiza la vista “mapa.html”.
    2. En la vista “mapa.html”, se importa la lógica para esta vista del script “map-scripts.js”
    3. Al hacerlo, primero se envía una petición GET a la API, en concreto al endpoint “obtener_refugios_geojson” cuya funcionalidad fue explicada en el apartado back end- endpoints.
    4. Tras recibir el GeoJson que contiene todos los refugios, se crea un mapa en Mapbox centrado en la Ciudad Autónoma de Buenos Aires, se agrega controles de navegación y un menú de direcciones.
    5. Luego, se generan las descripciones que se insertarán en los pop ups que aparecerán en pantalla al pasar el mouse sobre los puntos del mapa. Estas descripciones consisten en concatenaciones de propiedades de cada punto GeoJson, en concreto, sus propiedades nombre del refugio (title) y la dirección del mismo (address).
    6. Finalmente, se agrega una capa sobre el mapa que contenga todos los refugios presentes en el GeoJson.
- “**about_us”**
    1. La función about_us renderiza la vista “about_us.html”.
    2. En la vista “about_us.html”, se invita al usuario a ver los refugios, inscribirse como voluntario a un refugio o a agregar un refugio a la plataforma.
- “**feed”**
    1. La función feed renderiza la vista “feed.html”.
    2. En la vista “feed.html”, el usuario puede ver todos los refugios disponibles en la plataforma en formato card, con algunos detalles y sus imágenes.

Hipótesis y supuestos

Hipótesis:

- Existen centenares de refugios y centros de ayuda comunitaria con baja difusión entre sus comunidades.
- Existen miles de personas en necesidad de alimento, techo y abrigo.
- Existen cientos de personas con el tiempo y voluntad de colaborar con quienes más lo necesiten.

Supuestos:

- Un voluntario que busque colaborar con un refugio en Calle Solidaria sólo podrá hacerlo en un único refugio a la vez.
- Al registrar un refugio, el usuario proveerá una dirección completa que incluya:
  Calle y altura, Barrio/Municipio, Provincia, Código postal, País. Ej:
  Bonpland 1660, Palermo, Buenos Aires, 1440, Argentina.
- Al eliminar un refugio de la plataforma, los voluntarios también serán eliminados ya que dejarán de ser voluntarios de dicho refugio.

Referencias

1. Chequeado.com (14 de Marzo 2024). "Pobreza infantil: 7 de cada 10 niñas, niños y adolescentes en la Argentina son pobres". https://chequeado.com/el-explicador/pobreza-infantil-7-de-cada-10-ninas-ninos-y-adolescentes-en-la-argentina-son-pobres/
2. Documentación y referencias de API Mapbox GL JS https://docs.mapbox.com/mapbox-gl-js/api/
3. Documentación y referencias de API Mapbox Geocoding https://docs.mapbox.com/api/search/geocoding/
4. Documentación y referencias de API Mapbox Directions https://docs.mapbox.com/api/navigation/directions/
5. Lanzilotta, B (Abril de 2024). *Implementación de formularios*. Recuperado de https://www.youtube.com/watch?v=hcbIkmIfpPM&t=7299s
6. Lanzillotta, B (Mayo de 2024). *Conexión Base de Datos y Endpoints*. Recuperado de https://www.youtube.com/watch?v=xWO70rpbNi0&t=2786s

Anexo

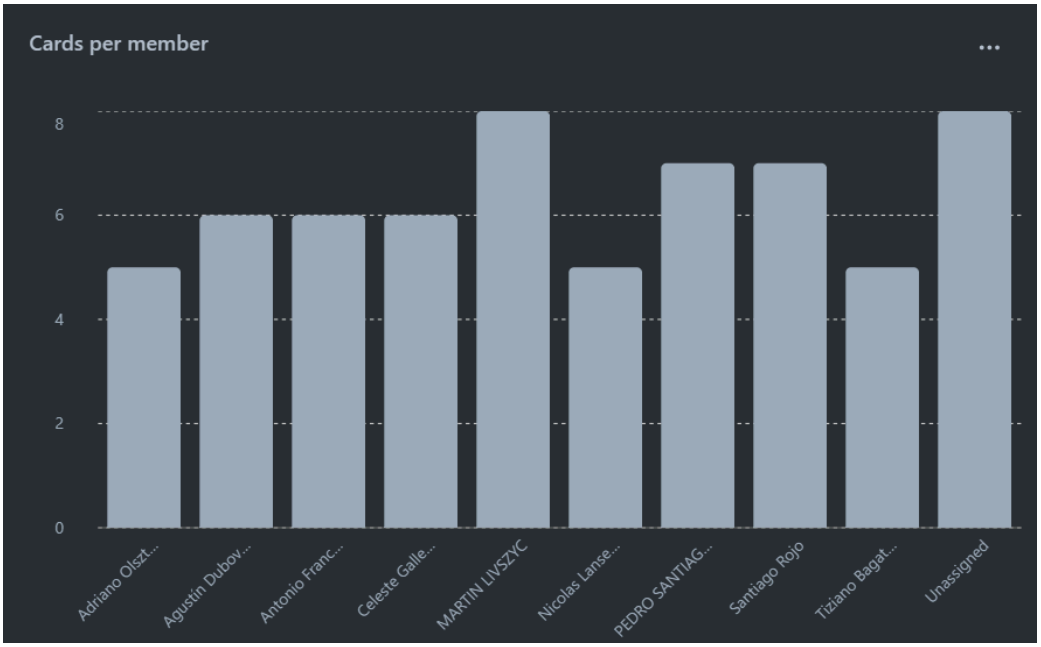

Figura 8. Este gráfico representa la cantidad de tareas asignadas por persona, teniendo todos un promedio de tareas entre 5 y 8. Vale recalcar la dificultad que implican algunas de las tareas, por lo cual cual dicho miembro puede tener más o menos asignadas.

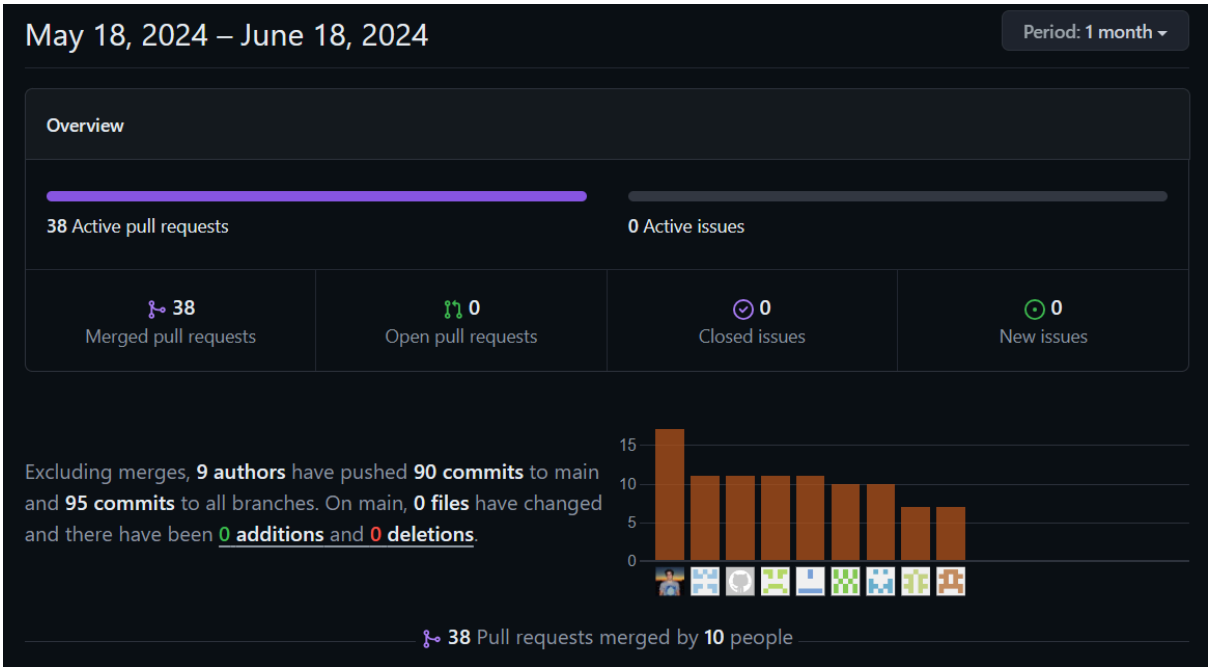

Figura 9. Este gráfico representa la actividad por persona en el repositorio de Github donde se trabajó el front end. Vale la pena recalcar que según los hábitos de cada miembro algunos commitearon más frecuentemente mientras que otros hacen commits menos frecuentes y más grandes, por lo cual cual dichos miembro puede tener más o menos commits al repositorio que no necesariamente representan la cantidad de trabajo que realizaron.

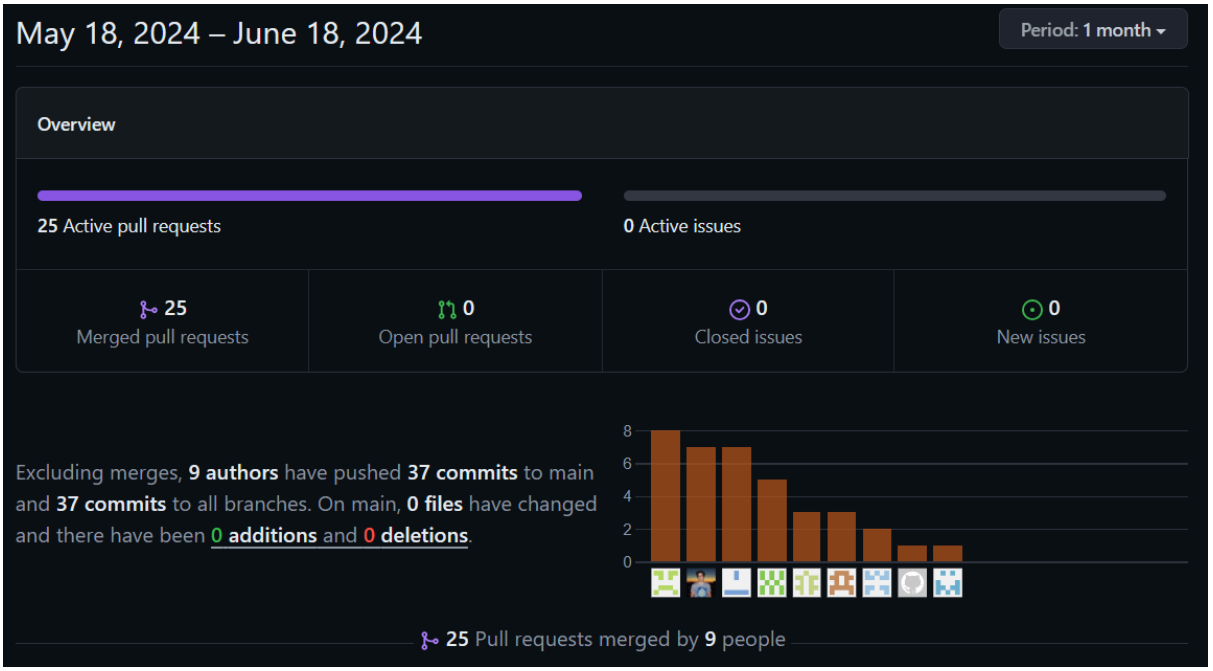

Figura 10. Este gráfico representa la actividad por persona en el repositorio de Github donde se trabajó el back-end de la plataforma, la API. Vale la pena recalcar que según los hábitos de cada miembro algunos commitearon más frecuentemente mientras que otros hacen commits menos frecuentes y más grandes, por lo cual cual dichos miembro puede tener más o menos commits al repositorio que no necesariamente representan la cantidad de trabajo que realizaron.