Informe de Desarrollo Full-Stack para SolidarianID

Desarrollo Full-Stack

Máster Universitario en Ingeniería del Software

Autores:

Hernán Salambay Roldán
Alejandro Montoya Toro
Pedro Nicolás Gomariz
Aurora Hervás López
Dongyue Yu

29 de Abril de 2025

Facultad de Informática
UMU

Contenidos

1 Introducción

En este documento se describe el desarrollo de la parte frontend de la plataforma SolidarianID, siguiendo los requisitos establecidos para el segundo cuatrimestre. Se ha implementado una aplicación en React y PWA que permite listar, buscar y visualizar comunidades, causas y acciones; gestionar el registro e inicio de sesión de usuarios; consultar el historial personal; solicitar la creación y unión a comunidades; apoyar causas solidarias y seguir a otros usuarios; así como gestionar notificaciones y validaciones por parte de los administradores.

Además, se han empleado tecnologías como Apollo Client para la integración con GraphQL y se han aplicado funcionalidades de Service Workers y notificaciones push para la experiencia PWA.

2 Frontend

En este apartado, se van a listar cada una de las funcionalidades requeridas en cuanto al uso de React, PWA y NEXT.js. Para cada una de ellas, se va a aportar una breve explicación así como fragmentos de código o capturas de la UI para poder mostrar claramente cómo se han implementado dentro de la plataforma.

A través del siguiente README, se puede visualizar cómo queda la aplicación final desplegada, sin necesidad de ejecutar el proyecto localmente.

2.1 Checklists de React

useState / useEffect (con return y dependencias)

Por un lado, el hook de estado useState se ha utilizado principalmente para almacenar valores en cada una de las páginas, como campos de formularios y el contenido de los alerts.

```
const navigate = useNavigate();
const [name, setName] = useState('');
const [page, setPage] = useState(1);
const [totalPages, setTotalPages] = useState(0);
const [communities, setCommunities] = useState<CommunityDetails[]>([]);
const [error, setError] = useState('');
```

Por otro lado, el hook de efecto se ha empleado en muchas partes del código. A continuación, se muestra un ejemplo para realizar búsquedas en el backend de forma automática al cambiar el estado de la página, como ocurre en la paginación con la búsqueda de comunidades, causas y acciones.

```
useEffect(() => {
  loadCommunities();
  // eslint-disable-next-line react-hooks/exhaustive-deps
}, [page]);
```

props

Las props se han usado para introducir datos en los componentes desde fuera de ellos. Un ejemplo claro se puede ver en el componente *ActionCard.*

```
export function ActionCard({
  id,
  title,
  description,
  achieved,
  target,
  status,
  type,
}: ActionDetails) {
```

Export default y nombrados

Hemos utilizado importación nombrada dentro del código, para evitar posibles conflictos de nombres y confusiones que pudieran dar lugar a errores.

Importación y uso de estilos

Aunque hemos utilizado principalmente React Bootstrap para dotar de estilo a la aplicación web, hemos utilizado el fichero index.css como hoja de estilos base.

```
:root {
  font-family: system-ui, Avenir, Helvetica, Arial, sans-serif;
  line-height: 1.5;
  font-weight: 400;

  color-scheme: light dark;
  color: rgba(255, 255, 255, 0.87);
  background-color: #242424;

  font-synthesis: none;
  text-rendering: optimizeLegibility;
  -webkit-font-smoothing: antialiased;
  -moz-osx-font-smoothing: grayscale;
}
```

```
import '../styles/index.css';
```

Manejo de eventos (y SyntheticEvent)

Se ha empleado manejo de eventos en aquellas funciones que representaban handlers de componentes, como por el ejemplo la función que gestiona el cambio de estado de un campo de un formulario.

```
<Form.Label>Password</Form.Label>
<Form.Control
  type="password"
  value={password}
  onChange={(e) => setPassword(e.target.value)}
  required
  placeholder="Enter your password"
  autoComplete="current-password"
/>
```

Compartiendo datos entre componentes (props)

Las props son la forma principal de pasar datos de un componente padre a un componente hijo. En nuestra aplicación se usan ampliamente para comunicar componentes. Ejemplo: Componente *ActionCard*

```tsx
// Definición de las props del componente
interface ActionCardProps {
  id: string;
  title: string;
  description: string;
  achieved: number;
  target: number;
  status: ActionStatusEnum;
  type: ActionTypeEnum;
}

// Componente que recibe props
export function ActionCard({
  id,
  title,
  description,
  achieved,
  target,
  status,
  type,
}: ActionCardProps) {
  const navigate = useNavigate();
```

Compartiendo datos con contexto global (useContext)

El hook *useContext* permite compartir datos entre componentes sin tener que pasar props explícitamente a través de cada nivel. Hemos usado este patrón principalmente para la autenticación en *AuthContext*:

```tsx
// Definición del contexto de autenticación
interface User {
  userId: string;
  firstName: string;
  lastName: string;
  roles: string;
  exp: number;
  token: string;
}

interface AuthContextType {
  user: User | null;
  isAuthenticated: boolean;
  login: (userData: User) => void;
  logout: () => void;
}
// Creación del contexto
const AuthContext = createContext<AuthContextType | undefined>(undefined);

// Proveedor del contexto que envuelve a la aplicación
export function AuthProvider({ children }: { children: ReactNode }) {
  const [user, setUser] = useState<User | null>(null);
  const [isAuthenticated, setIsAuthenticated] = useState(false);
```

```tsx
// Hook personalizado para acceder al contexto
export function useAuth() {
  const context = useContext(AuthContext);
  if (!context) throw new Error('useAuth must be used within an AuthProvider');
  return context;
}
```

Manipulando virtual DOM (useRef)

useRef es un hook de React que sirve para crear una referencia mutable a un elemento del DOM o a un valor que no causa re-renderizados al cambiar. El ejemplo típico es enfocar un input al cargar la página. En este caso hemos aplicado al input de email a la hora de hacer el login.

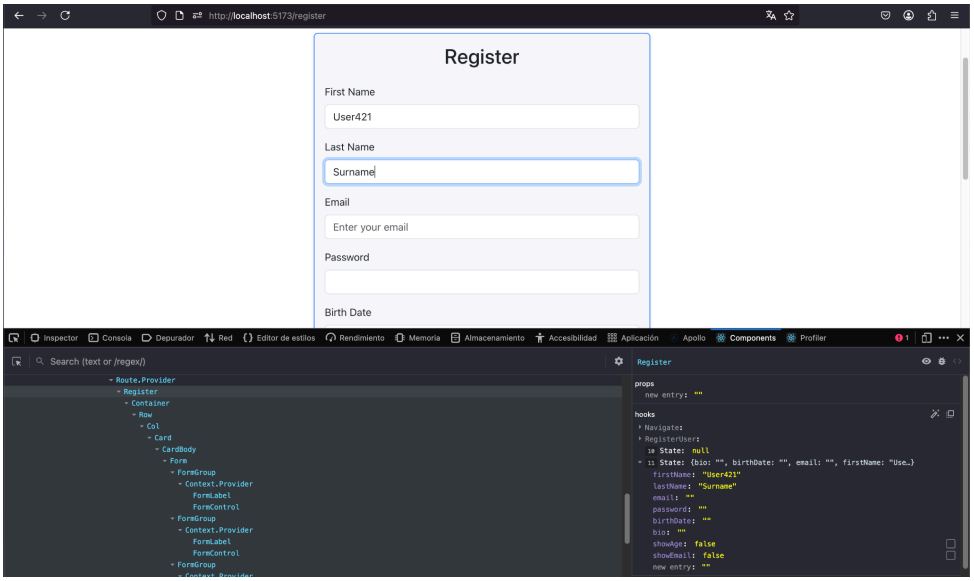
```
const emailRef = useRef<HTMLInputElement>(null);

useEffect(() => {
  emailRef.current?.focus();
}, []);
```

```
<Form.Control
  ref={emailRef}
  type="email"
  value={email}
  onChange={(e) => setEmail(e.target.value)}
  required
  placeholder="Enter your email"
  autoComplete="email"
/>
```

Debug con React Developer Tools

Principalmente se ha usado React Developer Tools para seguir la actualización de los estados y el flujo de los componentes, así como para localizar visualmente los componentes y facilitar la navegación hacia el código fuente, haciendo el proceso de depuración más sencillo e intuitivo.

Un ejemplo claro de uso es la revisión del estado en el formulario de registro: al modificar los datos, podemos observar en tiempo real cómo se actualizan los valores en el estado del componente, confirmando así que la lógica del formulario funciona correctamente.



Enrutamiento con React Router Dom (definir rutas , navegar Link)

Nuestra aplicación utiliza React Router DOM para gestionar la navegación entre páginas de forma fluida, sin recargar el navegador. En *routes.tsx*, definimos las rutas principales de la

aplicación usando el componente <*Routes*>, asociando cada path a su correspondiente componente de vista.

```
export function AppRoutes() {
  return (
    <Routes>
      {/* General */}
      <Route path="/" element={<Home />} />
      <Route path="/login" element={<Login />} />
      <Route path="/register" element={<Register />} />

      {/* User */}
      <Route path="/profile" element={<UserHistory />} />
      <Route path="/profile/:userId" element={<UserHistory />} />
      <Route path="/notifications" element={<Notifications />} />
```

En *CauseDetails.tsx*, usamos el componente <*Link*> para permitir al usuario navegar a los detalles de una comunidad relacionada, manteniendo una navegación rápida y sin recargas.

```
return (
  <Container className="py-5">
    {/* Header */}
    <Row className="justify-content-center text-center mb-4">
      <Col md={8}>
        <h5 className="text-muted mb-2">Cause Details</h5>
        <h2 className="fw-bold mb-3">{cause.title}</h2>
        {community && (
          <OverlayTrigger
            placement="top"
            overlay={<Tooltip id="tooltip-community">See Community Details</Tooltip>}
          >
            <Link
              to={`/communities/${community.id}`}
              className="d-inline-flex align-items-center gap-2 text-primary fw-semibold text-decoration-none"
            >
              <ArrowLeft size={18} />
              {community.name}
            </Link>
          </OverlayTrigger>
        )}
      </Col>
    </Row>
```

Enrutamiento con React Router Dom (useNavigate)

El hook *useNavigate* permite redirigir al usuario de forma programática, por ejemplo tras eventos como clics o envíos de formularios. En *CommunityCard*, al hacer clic en el botón, se usa navigate para redirigir al detalle de la comunidad correspondiente.

```
export function CommunityCard({ id, name, description }: CommunityCardProps) {
  const navigate = useNavigate();

  const handleClick = () => {
    if (!id) {
      console.error('No ID found for community:', { id, name, description });
      return;
    }
    navigate(`/communities/${id}`);
  };
```

Enrutamiento y uso de props: objeto to, state, update

La prop ***to*** define a dónde se dirige el enlace, a continuación se muestra un ejemplo de uso en *SolidarianNavbar.tsx*.

```
<Nav.Link as={NavLink} to="/communities">Communities</Nav.Link>
<Nav.Link as={NavLink} to="/causes">Causes</Nav.Link>
<Nav.Link as={NavLink} to="/actions">Actions</Nav.Link>
```

Enrutamiento con React Router Dom (NavLink)

NavLink funciona como *Link*, pero permite aplicar estilos automáticamente cuando la ruta está activa. En el ejemplo anterior se puede ver su uso.

Enrutamiento con React Router Dom (useLocation)

El hook *useLocation* permite acceder a la URL actual y usar su información en la lógica de los componentes. Hemos usado este hook en nuestro componente *SolidarianNavbar*, se usa location.pathname para detectar si estamos en la página de notificaciones y actualizar el contador de notificaciones en consecuencia.

```
export function SolidarianNavbar() {
  const { user, isAuthenticated, logout } = useAuth();
  const navigate = useNavigate();
  const location = useLocation();
  const [notificationCount, setNotificationCount] = useState<number>(0);
  const isNotificationsPage = location.pathname === '/notifications';

  //....

  useEffect(() => {
    if (isAuthenticated && user) {
      const savedCount = localStorage.getItem(`notifications_${user.userId}`);
      if (savedCount && !isNotificationsPage) {
        setNotificationCount(parseInt(savedCount, 10));
      } else if (isNotificationsPage) {
        setNotificationCount(0);
        localStorage.setItem(`notifications_${user.userId}`, '0');
      }
    }
  }, [isAuthenticated, user, isNotificationsPage]);
```

React REDUX : almacén, reductores y acciones

Redux es una librería de gestión de estado para aplicaciones JavaScript. Centraliza todo el estado en un único store y lo actualiza mediante acciones y reducers. Lo hemos aplicado para el login de usuario, permitiendo guardar y acceder a la sesión del usuario desde cualquier parte de la app.

```
// Create the auth slice
const authSlice = createSlice({
  name: 'auth',
  initialState,
  reducers: {
    // Login action
    login: (state, action: PayloadAction<User>) => {
      const userData = action.payload;
      const expirationTime = new Date();
      expirationTime.setHours(expirationTime.getHours() + 1);

      // Store user data in localStorage
      localStorage.setItem('user', JSON.stringify(userData));
      localStorage.setItem('token', userData.token);

      // Store user data in cookies -- for cross-domain authentication (Next.js frontend)
      Cookies.set('user', JSON.stringify(userData), {
        expires: expirationTime,
        path: '/',
        SameSite: 'None',
        Secure: true,
      });

      Cookies.set('token', userData.token, {
        expires: expirationTime,
        path: '/',
        SameSite: 'None',
        Secure: true,
      });
```

React Bootstrap

Para dotar de estilo a la aplicación, hemos hecho uso de React Bootstrap. Un ejemplo claro lo podemos ver en el componente *CauseCard*, donde hemos utilizado componentes como *Card* y *Button* de esta librería, los cuales disponen de un estilo predeterminado.

```
export function CauseCard({ id, title, description }: CauseDetails) {
  const navigate = useNavigate();

  const handleClick = () => {
    if (!id) {
      console.error('No ID found for cause:', { id, title, description });
      return;
    }
    navigate(`/causes/${id}`);
  };

  return (
    <Card className="h-100 shadow-sm" style={{ transition: 'transform 0.2s' }}>
      <Card.Body className="d-flex flex-column p-3">
        <Card.Title className="fs-5 mb-2">{title}</Card.Title>
        <Card.Text className="text-muted small flex-grow-1">{description}</Card.Text>
        <div className="d-grid mt-2">
          <Button variant="secondary" size="sm" onClick={handleClick}>
            See details
          </Button>
        </div>
      </Card.Body>
    </Card>
  );
}
```

Apollo Client para GraphQL: client, useQuery, inMemoryCache

Apollo Client es la biblioteca principal utilizada en SolidarianID para la comunicación con el backend a través de GraphQL. La implementación se encuentra principalmente en */frontend/src/lib/apolloClient.ts*.

Configuración del Cliente

Nuestra configuración de Apollo Client está diseñada para proporcionar una comunicación eficiente y centralizada con el servidor GraphQL, incorporando mecanismos para **consultas**, **mutaciones** y **suscripciones** en tiempo real:

Primero creamos un *httpLink* que configura la comunicación HTTP estándar para queries y mutaciones con el servidor GraphQL:

- Define el endpoint principal GraphQL (http://localhost:3000/graphql)
- Implementa un getter dinámico para la cabecera de autorización que obtiene el token JWT más reciente en cada solicitud
- Añade automáticamente el token en formato Bearer cuando está disponible
- Omite la cabecera de autorización cuando el usuario no está autenticado

```
// HTTP link with auth token
const httpLink = new HttpLink({
  uri: 'http://localhost:3000/graphql',
  headers: {
    get authorization() {
      const token = getToken();
      return token ? `Bearer ${token}` : '';
    },
  },
});
```

El *wsLink* establece la conexión WebSocket necesaria para las suscripciones en tiempo real:

- Configura el endpoint WebSocket (ws://localhost:3000/graphql)
- Habilita la reconexión automática cuando se pierde la conexión
- Mantiene la misma lógica de autorización que el enlace HTTP.

```
// WebSocket link for subscriptions using subscriptions-transport-ws
const wsLink = new WebSocketLink({
  uri: 'ws://localhost:3000/graphql',
  options: {
    reconnect: true,
    connectionParams: () => {
      const token = getToken();
      return {
        authorization: token ? `Bearer ${token}` : '',
      };
    },
  },
});
```

El *splitLink* implementa un sistema de enrutamiento de operaciones GraphQL:

- Analiza cada operación GraphQL para determinar su tipo utilizando getMainDefinition
- Dirige automáticamente las suscripciones al enlace WebSocket
- Envía las consultas y mutaciones al enlace HTTP

```
// Split links, so we do HTTP for queries and mutations, WS for subscriptions
const splitLink = split(
  ({ query }) => {
    const definition = getMainDefinition(query);
    return definition.kind === 'OperationDefinition' && definition.operation === 'subscription';
  },
  wsLink,
  httpLink
);
```

El *errorLink* implementa un sistema centralizado para interceptar y procesar todos los errores GraphQL y de red:

- Captura errores GraphQL específicos con información detallada sobre mensaje, ubicación y ruta
- Registra errores de red cuando ocurren problemas de conectividad
- Evita la duplicación de código de manejo de errores en componentes individuales

```
// Error handling link
const errorLink = onError(({ graphQLErrors, networkError }) => {
  if (graphQLErrors)
    graphQLErrors.forEach(({ message, locations, path }) =>
      console.error(`[GraphQL error]: Message: ${message}, Location: ${locations}, Path: ${path}`)
    );
  if (networkError) console.error(`[Network error]: ${networkError}`);
});
```

Finalmente, creamos y exportamos la instancia de *apolloClient* que será utilizada en toda la aplicación:

- Combina el *errorLink* con el *splitLink* en una cadena de enlaces.
- Configura la caché en memoria con políticas de normalización específicas.
- Establece opciones predeterminadas para diferentes tipos de operaciones.
- Proporciona una interfaz unificada para todas las comunicaciones GraphQL.

```
// Apollo Client instance
export const apolloClient = new ApolloClient({
  link: from([errorLink, splitLink]),
  cache,
  defaultOptions: {
    watchQuery: {
      fetchPolicy: 'cache-and-network',
      errorPolicy: 'all',
    },
    query: {
      fetchPolicy: 'cache-first',
      errorPolicy: 'all',
    },
    mutate: {
      errorPolicy: 'all',
    },
  },
});
```

Uso de useQuery

En React, combinamos Apollo Client con hooks para separar la lógica de datos de la interfaz de usuario. El siguiente ejemplo define *useUserById*, que consulta el perfil de un usuario:

```
/**
 * Hook to fetch a user by ID using GraphQL
 */
export const useUserById = (id: string) => {
  const { loading, error, data, refetch } = useQuery(GET_USER_BY_ID,
    variables: { id },
    skip: !id, // Skip the query if id is falsy
    fetchPolicy: 'cache-and-network', // Default fetch policy
  });

  return {
    loading,
    error,
    user: data?.user as UserProfile | undefined,
    refetch,
  };
};
```

A partir del hook *useUserById*, podemos construir hooks más específicos, como *useUserHistory*, que gestiona y sincroniza los datos del usuario en la interfaz. Este enfoque desacopla la consulta y el tratamiento de datos de la presentación, facilita el manejo de cambios en la API y promueve la reutilización en distintos componentes.

Apollo Client utiliza una caché en memoria para almacenar los resultados de las operaciones GraphQL. En nuestro caso se configura de la siguiente manera:

```
// Create cache with normalization via typePolicies
const cache = new InMemoryCache({
  typePolicies: {
    UserModel: {
      keyFields: ['id'],
    },
    CommunityModel: {
      keyFields: ['id'],
    },
    NotificationModel: {
      keyFields: ['id'],
    },
  },
});
```

Esta configuración usa las políticas de tipo (**typePolicies**) que normalizan los datos usando identificadores únicos, como *id* en los modelos *UserModel*, *CommunityModel* y *NotificationModel*. Esto permite almacenar respuestas GraphQL de forma estructurada, actualizar automáticamente la UI al cambiar los datos y controlar cómo se obtienen desde la caché o el backend. Así, se mejora el rendimiento y se mantiene una clara separación de responsabilidades en la aplicación.

Apollo Client para GraphQL: useMutation

El hook *useMutation* permite enviar modificaciones al servidor GraphQL y actualizar la caché local de forma controlada. A continuación, se define *useRegisterUser*, que registra un nuevo usuario:

```
/**
 * Hook for user registration mutation using GraphQL
 */
export const useRegisterUser = () => {
  const [mutate, { loading, error, data }] = useMutation(CREATE_USER, {
    update: (cache, { data }) => {
      if (data?.createUser) {
        const completeUserData = {
          ...data.createUser,
          // Add default values for fields that might be missing but are in the fragment
          age: data.createUser.age ?? 0,
          followersCount: data.createUser.followersCount ?? 0,
          followingCount: data.createUser.followingCount ?? 0,
        };

        cache.writeFragment({
          id: cache.identify(data.createUser),
          fragment: USER_EXTENDED_FRAGMENT,
          data: completeUserData,
        });

        cache.writeQuery({
          query: GET_USER_BY_ID,
          variables: { id: data.createUser.id },
          data: {
            user: completeUserData,
          },
        });
      }
    },
  });
```

Dentro del componente, se utiliza el hook para realizar la mutación y manejar la navegación tras el registro:

```
const handleSubmit = async (e: React.FormEvent) => {
  e.preventDefault();
  setRegistrationError(null);

  try {
    await registerUser(formData);
    navigate('/login');
  } catch (error) {
    setRegistrationError(error instanceof Error ? error.message : 'Error during registration');
  }
};
```

Esta implementación nos permite manejar la actualización de datos y la sincronización de la caché de forma explícita, mejorando la consistencia de la aplicación y la experiencia de usuario.

Apollo Client para GraphQL/cache: dataIdFromObject

En nuestro cliente Apollo se utiliza la funcionalidad de normalización mediante typePolicies (la evolución moderna de dataIdFromObject) para identificar objetos únicos en la caché. Esto se configura en la creación de la caché de Apollo:

```
// Create cache with normalization via typePolicies
const cache = new InMemoryCache({
  typePolicies: {
    UserModel: {
      keyFields: ['id'],
    },
    CommunityModel: {
      keyFields: ['id'],
    },
    NotificationModel: {
      keyFields: ['id'],
    },
  },
});
```

Esta configuración permite a Apollo Client normalizar entidades en la caché mediante identificadores únicos, establecer referencias entre objetos relacionados, mantener la coherencia de los datos en toda la aplicación y evitar duplicidades optimizando el uso de memoria. Por ejemplo, si un usuario se obtiene en varias consultas (perfil, comunidad, notificaciones), solo se almacena una copia en la caché, y cualquier actualización se refleja automáticamente en todos los componentes que lo utilizan.

Apollo Client para GraphQL/cache: uso de fragmentos

Los fragmentos en GraphQL permiten reutilizar partes de consultas. Nuestros fragmentos están definidos en */frontend/src/graphql/fragments.ts*:

```
export const USER_FRAGMENT = gql`
  fragment UserFields on UserModel {
    __typename
    id
    firstName
    lastName
    email
    bio
  }
`;

export const USER_EXTENDED_FRAGMENT = gql`
  fragment UserExtendedFields on UserModel {
    __typename
    id
    firstName
    lastName
    age
    email
    bio
    followersCount
    followingCount
  }
`;
```

Estos fragmentos se utilizan en consultas y mutaciones, lo que mejora la mantenibilidad y consistencia del código:

```
export const GET_USER_BY_ID = gql`
  query GetUser($id: ID!) {
    user(id: $id) {
      ...UserExtendedFields
    }
  }
  ${USER_EXTENDED_FRAGMENT}
`;
```

El uso de fragmentos nos aporta varios beneficios: evita la duplicación de campos (principio DRY), garantiza la consistencia de los datos en toda la aplicación, facilita la actualización de estructuras en un único lugar y optimiza la normalización de entidades en la caché. En nuestro caso, los fragmentos se aplican a distintas entidades como usuarios, comunidades y notificaciones, contribuyendo a mantener una base de código más ordenada y sostenible.

Apollo Client para GraphQL/cache: fetchPolicy

La propiedad *fetchPolicy* determina cómo Apollo Client interactúa con la caché. Se ha configuran diferentes políticas según el tipo de operación:

```
// Apollo Client instance
export const apolloClient = new ApolloClient({
  link: from([errorLink, splitLink]),
  cache,
  defaultOptions: {
    watchQuery: {
      fetchPolicy: 'cache-and-network',
      errorPolicy: 'all',
    },
    query: {
      fetchPolicy: 'cache-first',
      errorPolicy: 'all',
    },
    mutate: {
      errorPolicy: 'all',
    },
  },
});
```

Se configuran distintas políticas de caché para optimizar el rendimiento y la experiencia de usuario:

- **watchQuery**: 'cache-and-network': Devuelve datos de caché de forma inmediata y actualiza desde el servidor en segundo plano. Se usa en datos dinámicos como las notificaciones, ofreciéndonos velocidad sin sacrificar frescura.

- **query**: 'cache-first': Prioriza datos de caché y consulta la red solo si es necesario. Ideal para información estable como perfiles de usuario o detalles de comunidad, mejorando rendimiento y reduciendo consumo de red.

- **mutate**: 'errorPolicy': 'all': Permite recibir datos parciales y errores en mutaciones, facilitando un manejo más fino de fallos y actualizaciones optimistas en procesos como registro o creación de acciones.

Apollo Client para GraphQL/cache: update en useMutation

La opción **update** en *useMutation* permite actualizar la caché después de una mutación exitosa. Hemos usado esta funcionalidad a la hora de crear usuarios para mantener la caché actualizada con el estado del servidor:

```
/**
 * Hook for user registration mutation using GraphQL
 */
export const useRegisterUser = () => {
  const [mutate, { loading, error, data }] = useMutation(CREATE_USER, {
    update: (cache, { data }) => {
      if (data?.createUser) {
        const completeUserData = {
          ...data.createUser,
          // Add default values for fields that might be missing but are in the fragment
          age: data.createUser.age ?? 0,
          followersCount: data.createUser.followersCount ?? 0,
          followingCount: data.createUser.followingCount ?? 0,
        };

        cache.writeFragment({
          id: cache.identify(data.createUser),
          fragment: USER_EXTENDED_FRAGMENT,
          data: completeUserData,
        });

        cache.writeQuery({
          query: GET_USER_BY_ID,
          variables: { id: data.createUser.id },
          data: {
            user: completeUserData,
          },
        });
      }
    },
  });
```

Este enfoque permite actualizar la UI de forma inmediata, mantener la consistencia de los datos y reducir solicitudes innecesarias al servidor.

La función update recibe el objeto caché y el resultado de la mutación (data). En el ejemplo, se combinan *writeFragment* (para actualizar una entidad concreta) y *writeQuery* (para actualizar el resultado de una consulta relacionada), asegurando una sincronización completa tras crear un nuevo registro.

Apollo Client para GraphQL/cache: resetStore

El método *resetStore()* limpia toda la caché de Apollo Client y vuelve a ejecutar todas las consultas activas. Hemos optado por configurar este aspecto durante el cierre de sesión para garantizar que los datos del usuario anterior no persistan:

```
logout: (state) => {
  // Reset Apollo store
  apolloClient.resetStore().catch((err) => {
    console.error('Error resetting Apollo cache during logout:', err);
  });
```

A diferencia de *cache.reset()*, que simplemente borra la caché, *resetStore()* también vuelve a ejecutar todas las consultas activas, lo que asegura que la UI se actualice con los datos apropiados para el nuevo estado de autenticación.

Adicionalmente, se ha empleado el hook *useSubscription* para mantener actualizado en tiempo real el contador de nuevas notificaciones en el navbar, reflejando automáticamente cada inserción de notificaciones dirigida al usuario.

2.2 Checklists de PWA

PWA en React con Workbox

Para convertir la aplicación SolidarianID en una PWA, se ha utilizado la biblioteca Workbox. La integración se ha realizado a través del plugin *vite-plugin-pwa*. El **service-worker.js** se encuentra definido en la carpeta /src del proyecto React y se inyecta en el proceso de build mediante la estrategia *injectManifest* de Workbox. Esta configuración se puede ver reflejada en el archivo *vite.config.ts*, donde también se define el **manifest** web que cumple con los requisitos para el reconocimiento como PWA.

```
VitePWA({
  srcDir: 'src',
  filename: 'service-worker.js',
  strategies: 'injectManifest',
  injectManifest: {
    swSrc: 'src/service-worker.js',
    swDest: 'dist/service-worker.js',
    maximumFileSizeToCacheInBytes: 3000000,
  },
  registerType: 'autoUpdate',
  devOptions: {
    enabled: true,
    type: 'module',
  },
  manifest: {
    name: 'SolidarianID',
    short_name: 'SolidarianID',
    start_url: '/',
    display: 'standalone',
    background_color: '#ffffff',
    theme_color: '#1a73e8',
    icons: [
```

El registro del service worker se realiza a través del componente *ServiceWorkerRegistration.tsx*, que se llama en el archivo *main.tsx*:

```
const ServiceWorkerRegistration = () => {
  useEffect(() => {
    if ('serviceWorker' in navigator) {
      const wb = new Workbox('/service-worker.js');

      wb.register()
        .then((registration) => {
          console.log('Service Worker registrado con éxito:', registration);
        })
        .catch((error) => {
          console.error('Error al registrar el Service Worker:', error);
        });

      wb.addEventListener('activated', (event) => {
        if (event.isUpdate) {
          console.log('¡Nuevo Service Worker activado!');
          window.location.reload();
        }
      });
```

Estrategias de caché en Workbox: Precaching

Utilizando la función *precacheAndRoute* de Workbox, se precachean los archivos estáticos como scripts, imágenes y ficheros HTML generados durante la compilación (en la carpeta dist). Esto se configura en el service-worker.

```
import { precacheAndRoute } from 'workbox-precaching';

precacheAndRoute(self.__WB_MANIFEST);
```

Estrategias de caché en Workbox: Runtime Caching

```
import { CacheFirst, NetworkFirst } from 'workbox-strategies';
```

Para los recursos estáticos como estilos, imágenes y scripts, se ha implementado la estrategia **Cache First** de workbox, que prioriza el contenido que ya está almacenado en caché.

```
registerRoute(
  ({ request }) =>
    request.destination === 'image' ||
    request.destination === 'style' ||
    request.destination === 'script',
  new CacheFirst({
    cacheName: 'static-resources',
    plugins: [
      new ExpirationPlugin({
        maxEntries: 50,
        maxAgeSeconds: 30 * 24 * 60 * 60, // 30 Days
      }),
    ],
  })
);
```

Para las peticiones realizadas a la API se implementa la estrategia **Network First.** Primero se intenta obtener la respuesta de la red, y si falla o no hay conexión, se sirve (si existe) una copia almacenada en caché. Ambas estrategias se implementan en el service-worker:

```
registerRoute(
  ({ url }) => url.pathname.startsWith('/api/'),
  new NetworkFirst({
    cacheName: 'api-resources',
    plugins: [
      new ExpirationPlugin({
        maxEntries: 50,
        maxAgeSeconds: 5 * 60, // 5 Minutes
      }),
    ],
  })
);
```

Notificaciones Push

Se implementan notificaciones push definiendo el evento en el service-worker para mostrar una notificación con el contenido del mensaje. Además al hacer clic sobre ella se abre una ventana con la ruta */notifications* de la aplicación.

```
self.addEventListener('push', (event) => {
  const payload = event.data?.text() ?? 'No payload';
  event.waitUntil(
    self.registration.showNotification('New notification from SolidarianID', {
      body: payload,
      data: { url: '/notifications' },
    })
  );
});

self.addEventListener('notificationclick', (event) => {
  const url = event.notification.data.url;
  event.waitUntil(clients.openWindow(url));
});
```

Para servir las notificaciones se cuenta con el proyecto **server_push,** incluido en el repositorio, que define los endpoints *vapidPublicKey*, *register* y *sendNotification*, entre otros, para permitir al usuario suscribirse y enviar notificaciones desde el backend.

Por otra parte, cuando un usuario inicia sesión se llama a la funcion *enableNotifications()* del servicio */services/push-notification.service.ts*, que se encarga de solicitar permiso al usuario para activar las notificaciones y suscribirse al server_push para escucharlas.

```
export async function enableNotifications() {
  if ('Notification' in window && 'PushManager' in window) {
    const permission = await Notification.requestPermission();
    if (permission === 'granted') {
      try {
        const registration = await navigator.serviceWorker.ready;
        const subscription = await subscribeUserToPushManager(registration);

        const userId = JSON.parse(localStorage.getItem('user') || '{}').userId;

        await registerSubscriptionOnServer(subscription, userId);
        return true;
```

2.2 Checklists de NEXT

En este apartado describimos los elementos de Next.js que hemos implementado en nuestro proyecto, detallando su uso y aplicación práctica.

NEXT.js tipos de renderizado: SSR

El tipo de renderizado SSR (Server-Side Rendering) se usa en la vista de *[causeId].tsx*, vista encargada de mostrar los detalles de una causa. Se consigue este tipo de renderizado a través

de la función *getServerSideProps* que se ejecuta en el servidor en cada solicitud. Se usa para obtener datos dinámicos (causa, comunidad, acciones, etc.) antes de renderizar la página.

```
export const getServerSideProps: GetServerSideProps = async (context) => {
  const causeId = context.params?.causeId;

  if (!causeId) {
    return {
      notFound: true,
    };
  }

  const cookies = context.req.cookies;
  const userData = cookies.user ? JSON.parse(cookies.user) : null;

  // Fetch cause data
  const causeResponse = await fetch(`http://localhost:3000/api/v1/causes/${causeId}`, {
    method: 'GET',
    headers: {
      'Content-Type': 'application/json',
    },
  });

  if (!causeResponse.ok) {
    return {
      notFound: true,
    };
  }
```

NEXT.js tipos de renderizado: SSR con CSR

Este renderizado es una mezcla de SSR con CSR (Client-Side Rendering) que usa SSR para obtener los datos iniciales y luego hace *fetch* desde el cliente para cargar acciones relacionadas en el *useEffect*. En este caso lo hemos usado cuando se cambia de página en la paginación de acciones de una causa.

```
useEffect(() => {
  const fetchActions = async () => {
    setIsLoadingActions(true);
    try {
      const res = await fetch(`http://localhost:3000/api/v1/causes/${causeId}/actions?page=${currentPage}&limit=${limit}`);

      if (!res.ok) throw new Error('Error fetching actions');

      const data = await res.json();
      setTotalPages(data.meta.totalPages);
```

NEXT.js enrutamiento con Pages Router

Es el sistema tradicional de rutas en Next.js, basado en carpetas y archivos dentro de *pages/*. La página de detalles de una causa está en la ruta *pages/causes/[causeId].tsx* que se convierte en la URL */causes/:causeId* y se puede acceder al parámetro desde router.query

```
const router = useRouter();
const { causeId } = router.query;
```

NEXT.js enrutamiento con App Router

Es un sistema de enrutamiento nuevo que reemplaza al Pages Router. En lugar de usar *pages/* usa la carpeta *app/* para definir las rutas. Como hemos usado el enrutamiento con Pages Router no podemos aplicar este tipo de enrutamiento en el mismo proyecto, por lo tanto no hemos llegado a implementarlo.

NEXT.js enrutamiento paralelo

Como el enrutamiento paralelo (*@parallel, @slot*, etc.) es exclusivo del *app/* router en Next.js y nuestro proyecto está basado en *pages/* no ha sido posible aplicarlo.

NEXT.js React Server Components

Los React Server Components (RSC) son una funcionalidad exclusiva del sistema de rutas *AppRouter* en Next.js que permite renderizar componentes directamente en el servidor sin enviar su lógica al navegador, optimizando así el rendimiento. En este proyecto no se usan porque está basado en el sistema tradicional de rutas pages/, donde todos los componentes funcionan como Client Components por defecto.

NEXT.js Optimización Lazy Load: imágenes

Next.js carga imágenes solo cuando están cerca del viewport, mejorando el rendimiento inicial. Lo hemos usado para cargar las imágenes de los ODS en la vista de detalle de una causa.

```
<div className='image-container'>
  <Image
        src={odsImages[ods.id].src}
        alt={ods.title}
        className="img-fluid image-border"
        width={100}
        height={100}
        loading="lazy"
      />
</div>
```

NEXT.js Optimización Lazy Load: componentes

Next.js permite cargar componentes sólo cuando realmente se necesitan, mejorando el tiempo de carga. En este caso, por la falta de vista implementada, hemos aplicado en el *SolidarianNavbar* dentro de *_app.tsx*

```
const SolidarianNavbar = dynamic(() => import('@/components/SolidarianNavbar'), {
  ssr: false,
});
```

NEXT.js Optimización Lazy Load: importación dinámica

Se aplicó explícitamente en el mismo fragmento de código anterior, con *ssr: false* evitando que el navbar se cargue en el SSR; solo se carga en el cliente cuando se necesita. Es un poco raro, pero al tener pocas vistas, sólo hemos podido aplicar en ese sitio.

NEXT.js Optimización Streaming con suspense

React. Suspense permite mostrar contenido alternativo (como "loading...") mientras se carga algo asincrónico. Este caso lo hemos aplicado en el navbar. El fallback se muestra mientras *SolidarianNavbar* está cargando. Aunque no es streaming del lado del servidor, es Suspense del lado del cliente.

```
<Suspense fallback={<p>Loading Navbar...</p>}>
  <SolidarianNavbar />
</Suspense>
```

3 Backend

GraphQl

Se ha implementado un adaptador GraphQL en el microservicio de API Gateway como mecanismo unificador de los microservicios del backend, proporcionando una interfaz de comunicación más flexible y eficiente.

El sistema permite consultar el perfil de usuario, con campos como *followersCount* y *followingCount* calculados dinámicamente, y los detalles de comunidad, resolviendo relaciones como el admin de forma condicional, optimizando así el número de consultas necesarias desde el frontend para construir el perfil y la información asociada a las comunidades. Además, soporta la creación de usuarios mediante mutaciones y las suscripciones a notificaciones en tiempo real.

La arquitectura incorpora un manejo de errores centralizado mediante GraphQLGeneralExceptionFilter, mapeando las excepciones en respuestas comprensibles para el cliente. La comunicación de eventos entre microservicios se realiza a través de Kafka, permitiendo al API Gateway publicar estos eventos mediante las suscripciones de GraphQL.

Correcciones respecto a versiones anteriores.

- **Rediseño del seguimiento de usuarios**: Se ha sustituido el modelo anterior, basado en arrays embebidos, por un módulo dedicado. Este nuevo módulo utiliza una entidad Follower para gestionar las relaciones, incorporando consultas optimizadas, paginación y metodologías que favorecen una mayor escalabilidad.

- **Optimización del sistema de notificaciones**: Se ha establecido una conexión entre los eventos de actividad (HistoryEntry) y las notificaciones. Esto permite una distribución más selectiva y comprensible a seguidores, administradores y entidades relacionadas. Para la gestión de grandes volúmenes, se ha implementado el procesamiento por lotes.

- **Mejora del modelado del historial de usuario**: El modelo de historial se ha adaptado para cubrir diversos casos de uso, registrando las actividades de manera más precisa y simplificada. Adicionalmente, se han añadido nuevos tipos de notificaciones basados en ActivityType, como JOIN_COMMUNITY_REQUEST_SENT y COMMUNITY_CREATION_REQUEST_SENT. Estas notificaciones cuentan con un tratamiento específico, integración con push notifications y publicación en tiempo real a través de WebSockets y suscripciones GraphQL.

- **Incorporación de gestión de errores**: Se ha añadido un método `handleError` al controlador del microservicio de comunidades con el fin de centralizar la gestión de errores y prevenir la duplicación de código.

- **Autenticación OAuth2**: Para la autenticación con OAuth2, se integró un nuevo campo en la base de datos. Este campo almacena el identificador único de Github, que ahora constituye el método de inicio de sesión correcto para esta plataforma.

4 Puesta en marcha

En este apartado se detallan los pasos necesarios para poner en funcionamiento el proyecto completo SolidarianID, incluyendo tanto el frontend como el backend.

1. **Acceder al repositorio oficial del proyecto disponible en GitHub.**
2. **Revisión del archivo README.** Contiene un diagrama de arquitectura de contenedores que proporciona una visión general, así como los comandos principales necesarios para levantar la aplicación en entorno de producción:
    a. Para crear la red necesaria en el entorno de producción: *make create-network*
    b. Para iniciar la aplicación en entorno de producción: *make run-prod*
3. **Cargar datos de prueba.** Una vez que la infraestructura esté en funcionamiento se pueden cargar datos de prueba a través de un script de python que realiza peticiones a la API del backend.
    a. cd scripts/test_data_loader
    b. python3 test_data_loader.py
4. **Acceso a la aplicación.** Se puede acceder a la aplicación a través del navegador en la URL http://localhost:5173 con las credenciales: *admin@admin.com:123456Test**.

Para probar la parte de frontend integrando la parte de NEXT es necesario lanzar la aplicación de la otra forma. Dentro del CauseCard.tsx hay que modificar la URL:

```
export function CauseCard({ id, title, description }: CauseCardProps) {
  const navigate = useNavigate();

  const handleClick = () => {
    if (!id) {
      console.error('No ID found for cause:', { id, title, description });
      return;
    }
    window.location.href = `http://localhost:3006/causes/${id}`;
    //navigate(`/causes/${id}`);
  };
```

y desde la carpeta de backend hay que ejecutar los siguientes comandos:

- *make run-dev* para lanzar los contenedores
- cargar los datos de pruebas como se indica anteriormente
- lanzar cada uno de los microservicios por separado:
    - *npm run start:dev:api-gateway*
    - *npm run start:dev:users-ms*
    - *npm run start:dev:communities-ms*

y desde la carpeta de frontend y frontend-next se ejecuta el comando ***npm run dev***.

5 Bibliografía

- Documentación oficial de NestJS: https://docs.nestjs.com/.
- Documentación oficial de React: https://es.react.dev/.
- Documentación oficial de Boostrap: https://react-bootstrap.netlify.app/docs/components/accordion.
- Documentación oficial de Next.js: https://nextjs.org/docs.
- Transparencias de la asignatura Desarrollo Full-Stack.