MANUAL DE PROGRAMADOR

Aplicación SITApp

Grupo de Investigación Aplicada en Sistemas

TABLA DE CONTENIDO

TABLA DE FIGURAS

MANUAL DE PROGRAMADOR SITAPP

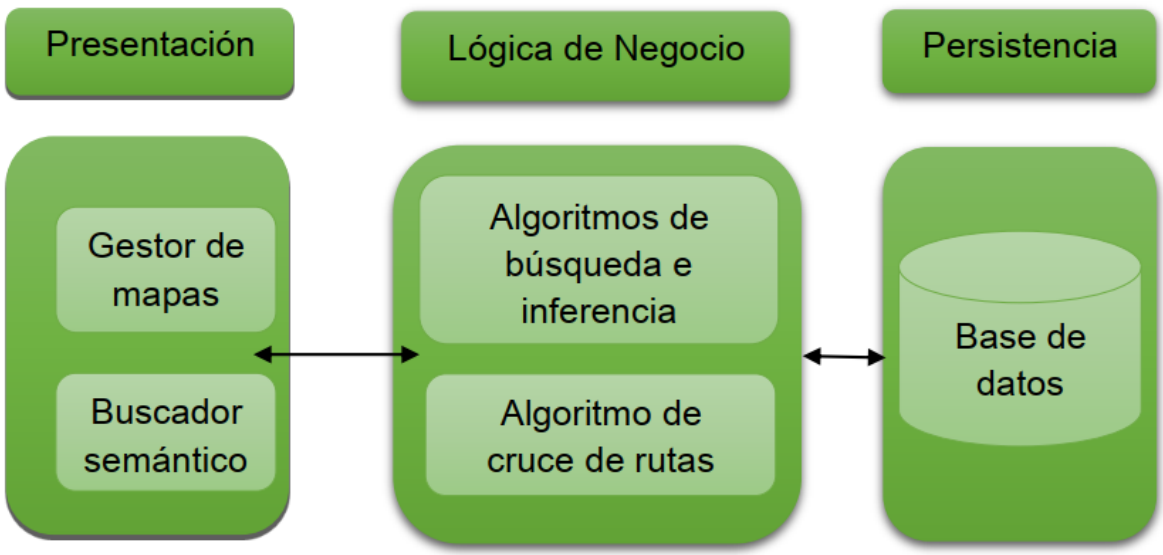
El presente documento es una guía para los programadores que hacen uso de SITApp Android y SITApp Web Service, es un manual práctico para la compresión y descripción de las principales funcionalidades que tiene este aplicativo móvil y su Web Service.

La construcción del aplicativo fue realizada bajo el lenguaje de programación Java, el cual es el lenguaje nativo de programación para Android, quien hace uso a su vez del lenguaje XML para los patrones de diseño de las interfaces gráficas de la aplicación. Se usó el IDE de desarrollo oficial Android Studio en el sistema operativo Debian 8, para el manejo de mapas se usó la librería Mapbox SDK, para la interacción de SITApp con el Web Service se usó la librería Retrofit.

La construcción del Web Service fue realizada bajo el lenguaje de programación Java Enterprise Edition 7 (Java EE7), haciendo uso del IDE Netbeans 8.1. Este aplicativo hace uso de la tecnología RESTful que utiliza los métodos HTTP y permite transferir archivos JavaScript Object Notation (JSON) para establecer comunicación entre el aplicativo en Android y la base de datos. Para el soporte de conexión y consultas hacia la ontología de SITApp se usó la librería Jena 2.7, adicionalmente la librería Lucene ayuda al Web Service a eliminar palabras muertas en las búsquedas del usuario y hacerlas más eficientes.

1. ARQUITECTURA

Figura 1. Arquitectura SITApp



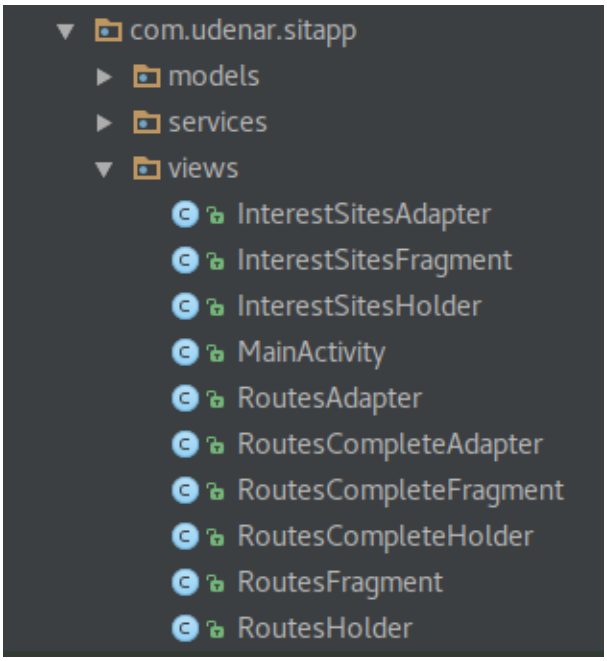
Fuente: Esta investigación.

1.1. Capa de presentación

En esta capa se encuentran albergados todos los elementos de presentación gráficos para el usuario haciendo que los procesos que realiza el servidor para gestionar toda la información sean transparentes y disminuyendo cargas en el procesamiento de datos desde el lado del cliente, los elementos que le permiten interactuar con la aplicación y sus diferentes opciones son:

- Los gestores de mapas, que permiten visualizar toda la información georreferenciada acerca del SETP almacenada en el servidor desplegando esta información de distintas maneras de acuerdo a los distintos tipos de consultas que el usuario realice en su momento, mostrando las geometrías de los recorridos las rutas, los marcadores que representan a los paraderos o los sitios de interés de la ciudad de Pasto.
- El buscador semántico encargado de gestionar las consultas del usuario y realizar las peticiones hacia el Web Service, este buscador se comunica con la capa de presentación para mostrar los resultados de la búsqueda sin importar los errores de ortografía del usuario.

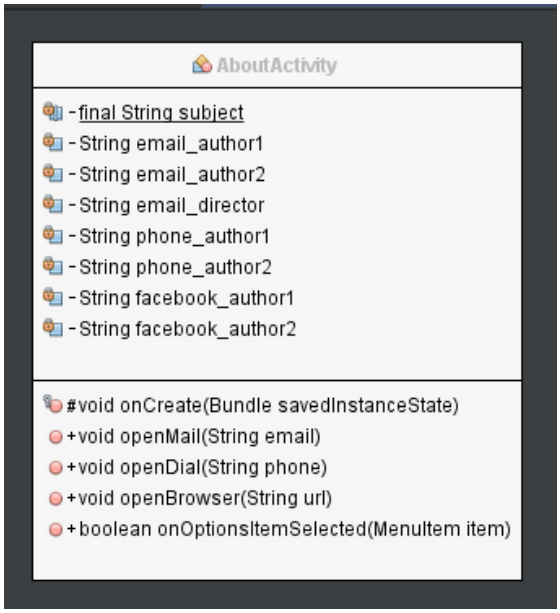
Figura 2. Clases del paquete views de SITApp



Fuente: Esta investigación.

A continuación se describen las clases que componen este paquete:

Figura 3. Clase AboutActivity.java



Fuente: Esta investigación.

Esta clase es la encargada de mostrar el activity que posee toda la información de contacto de los desarrolladores de la aplicación.

Figura 4. Clase InterestSitesAdapter.java

Fuente: Esta investigación.

Clase que recibe y almacena el listado de sitios de interés.

Figura 5. Clase InterestSitesFragment.java

Fuente: Esta investigación.

Clase encargada de crear el fragmento y hacerlo visible en la pantalla.

Figura 6. Clase InterestSitesHolder.java

Fuente: Esta investigación.

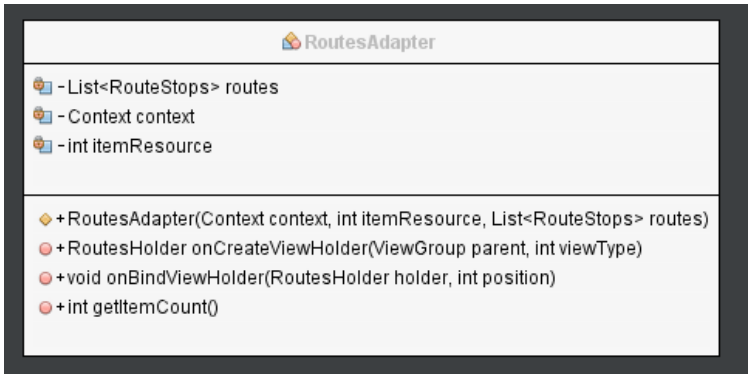
Clase que representa cada uno de los sitios de interés de la lista y se encarga de realizar las acciones necesarias cuando el sitio de interés es seleccionado por el usuario.

Figura 7. Clase MainActivity.java

Fuente: Esta investigación.

Clase principal que se encarga de gestionar y mostrar el mapa y las listas de rutas y sitios de interés, posee cada uno de los métodos necesarios para establecer la comunicación entre SITApp y el Web Service.

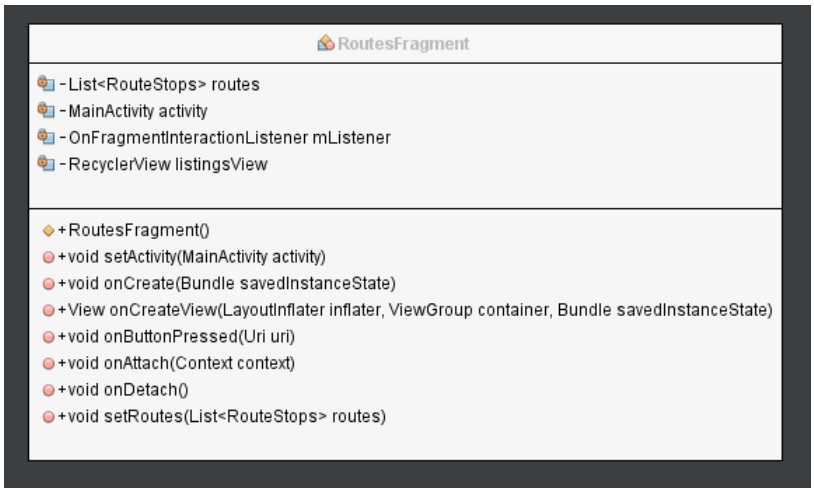
Figura 8. Clase RoutesAdapter.java



Fuente: Esta investigación.

Clase que recibe y almacena el listado de las rutas con el paradero de origen y destino.

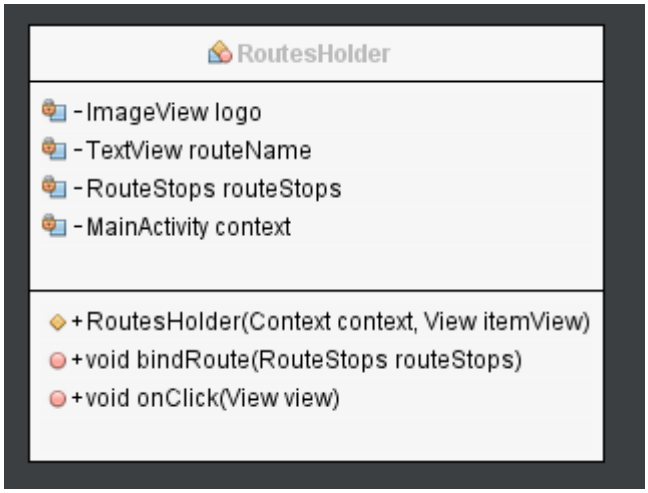
Figura 9. Clase RoutesFragment.java



Fuente: Esta investigación.

Clase encargada de crear el fragmento y hacerlo visible en la pantalla.

Figura 10. Clase RoutesHolder.java



Fuente: Esta investigación.

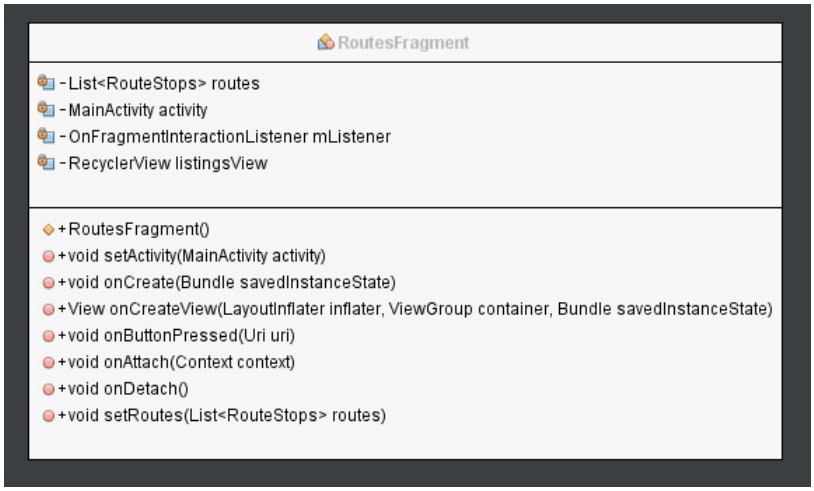
Clase que representa cada una de las rutas en la lista y se encarga de realizar las acciones necesarias cuando la ruta es seleccionada por el usuario.

Figura 11. Clase RoutesCompleteAdapter.java

Fuente: Esta investigación.

Clase que recibe y almacena el listado de las rutas con todos sus paraderos asociados.

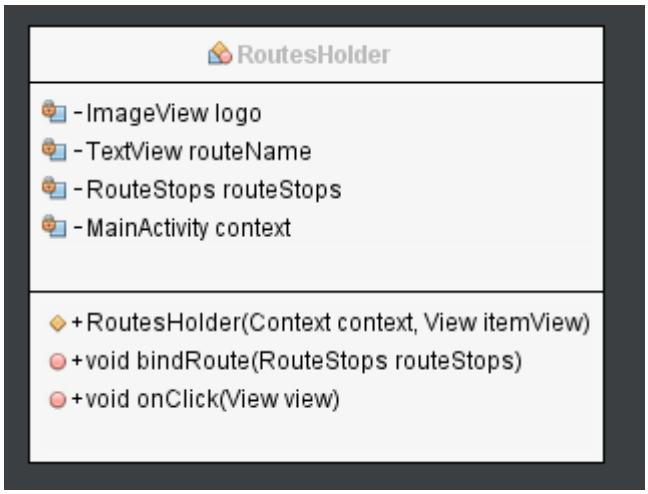
Figura 12. Clase RoutesCompleteFragment.java



Fuente: Esta investigación.

Clase encargada de crear el fragmento y hacerlo visible en la pantalla.

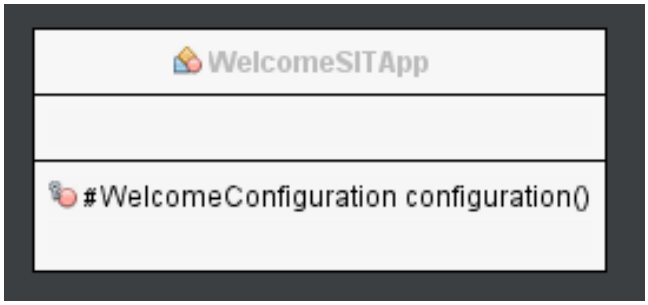
Figura 13. Clase RoutesCompleteHolder.java



Fuente: Esta investigación.

Clase que representa cada una de las rutas en la lista y se encarga de realizar las acciones necesarias cuando la ruta es seleccionada por el usuario.

Figura 14. Clase WelcomeSITApp.java



Fuente: Esta investigación.

Esta clase es la encargada de desplegar el menú de bienvenida de la aplicación.

1.2. Lógica de negocio

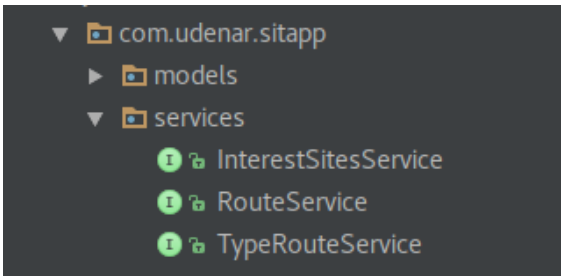
En esta capa se encuentran albergados los distintos algoritmos que se encargan de gestionar las distintas peticiones recibidas por la capa de presentación, aquí se lleva a lugar la tarea de procesamiento y gestión de toda la información georreferenciada mediante:

- Algoritmos de búsqueda e inferencia, encargados de gestionar consultas complejas realizando inferencias semánticamente haciendo uso de la ontología.
- Algoritmo de cruce de rutas, encargado de realizar la búsqueda de la o las rutas que llevan al usuario desde un punto geográfico "A" hasta un punto geográfico "B".

Además se encuentran los procedimientos encargados de la comunicación entre SITApp y el Web Service.

Esta capa está compuesta por los paquetes "services" de SITApp y "conection", "queries" y "service" del Web Serice. A continuación se describen cada uno de ellos y sus clases.

Figura 15. Clases del paquete services de SITApp



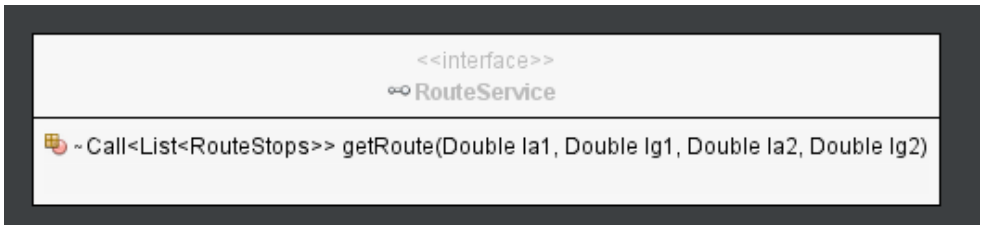
Fuente: Esta investigación.

Figura 16. Interface InterestSitesService.java

Fuente: Esta investigación.

Interface encargada de controlar el método de envío de información hacia el Web service, lo realiza mediante el método GET.

Figura 17. Interface RouteService.java



Fuente: Esta investigación.

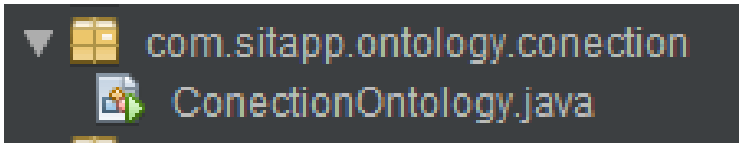
Interface encargada de controlar el método de envío de información hacia el Web service, lo realiza mediante el método GET.

Figura 18. Interface TypeRouteService.java

Fuente: Esta investigación.

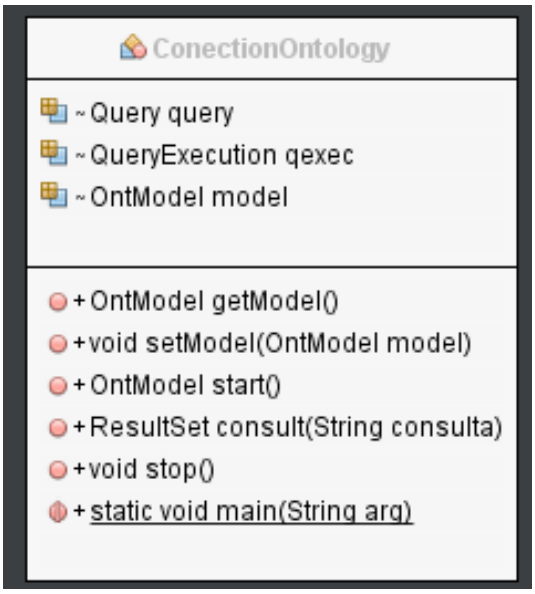
Interface encargada de controlar el método de envío de información hacia el Web service, lo realiza mediante el método GET.

Figura 19. Clases del paquete ontology.conection del SITApp Web Service



Fuente: Esta investigación.

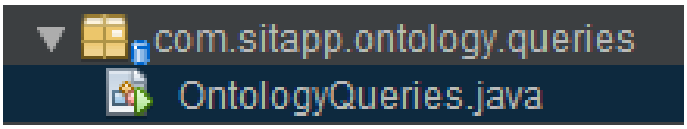
Figura 20. Clase ConectionOntology.java



Fuente: Esta investigación.

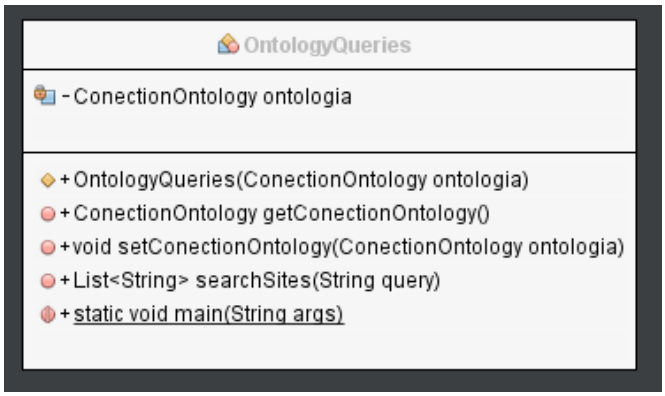
Clase encargada de establecer la conexión con la ontología que se encuentra almacenada en el servidor.

Figura 21. Clases del paquete ontolgy.queries de SITApp Web Service



Fuente: Esta investigación.

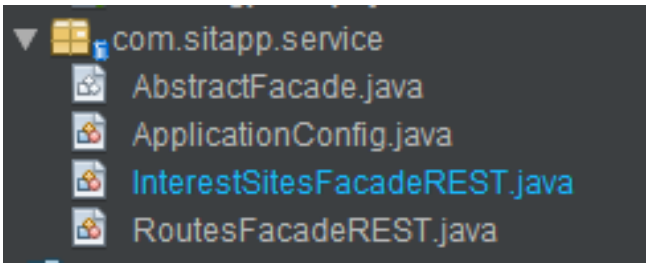
Figura 22. Clase OntologyQuieries.java



Fuente: Esta investigación.

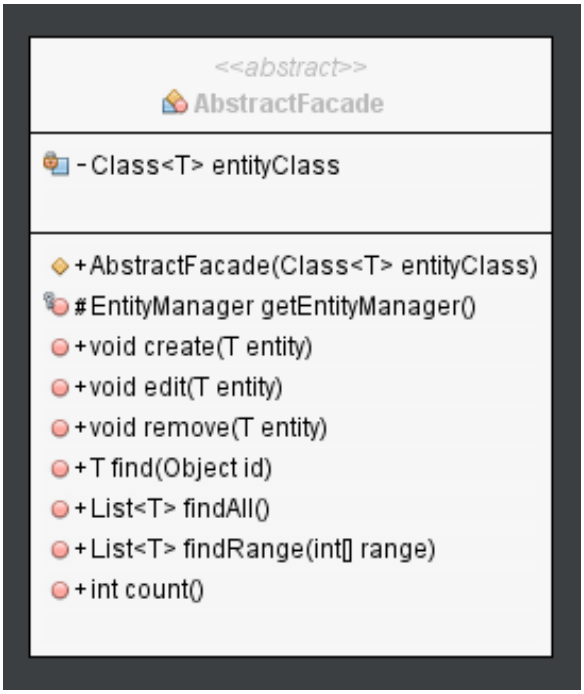
Clase que contiene las consultas SPARQL necesarias para acceder a los datos almacenados en la ontología.

Figura 23. Clases del paquete service de SITApp Web Service



Fuente: Esta investigación.

Figura 24. Clase AbstractFacade.java



Fuente: Esta investigación.

Clase abstracta de la cual heredan los servicios del Web Service, contiene métodos necesarios para la comunicación entre el modelo de la base de datos que se encuentra en la capa de persistencia.

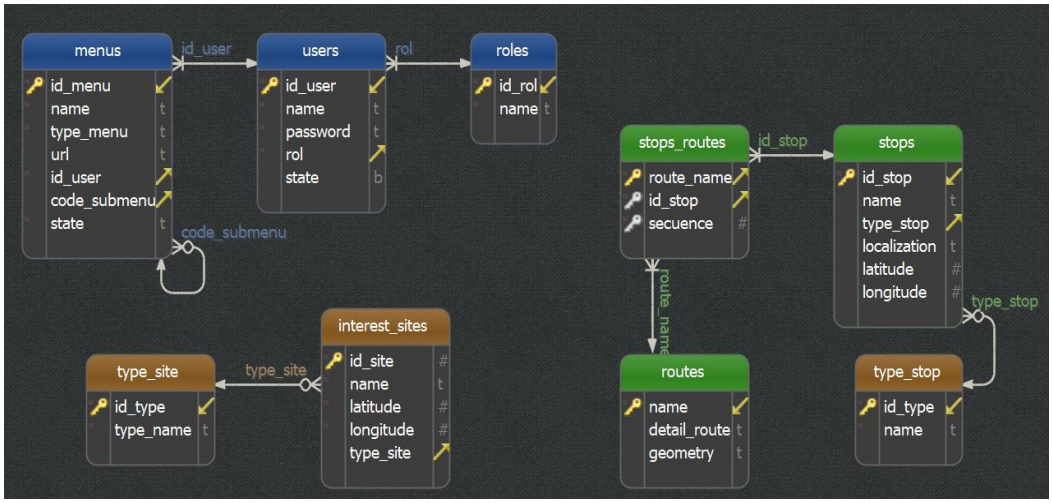
1.3. Capa de persistencia

En esta capa se encuentra la base de datos de SITApp, que alberga toda la información georreferenciada acerca de las rutas y paraderos del SETP y los sitios de interés más

relevantes de la ciudad de Pasto, aquí también se encuentran las clases encargadas de mapear este modelo tanto en SITApp como en el Web Service.

A continuación se muestra el diagrama entidad relación de la base de datos de SITApp.

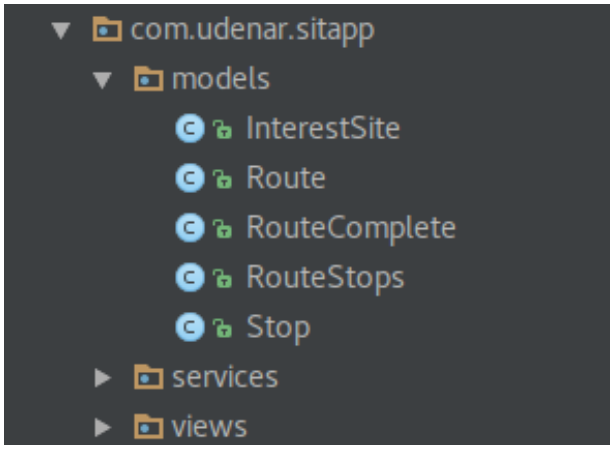
Figura 25. Diagrama entidad relación de SITApp.



Fuente: Esta investigación.

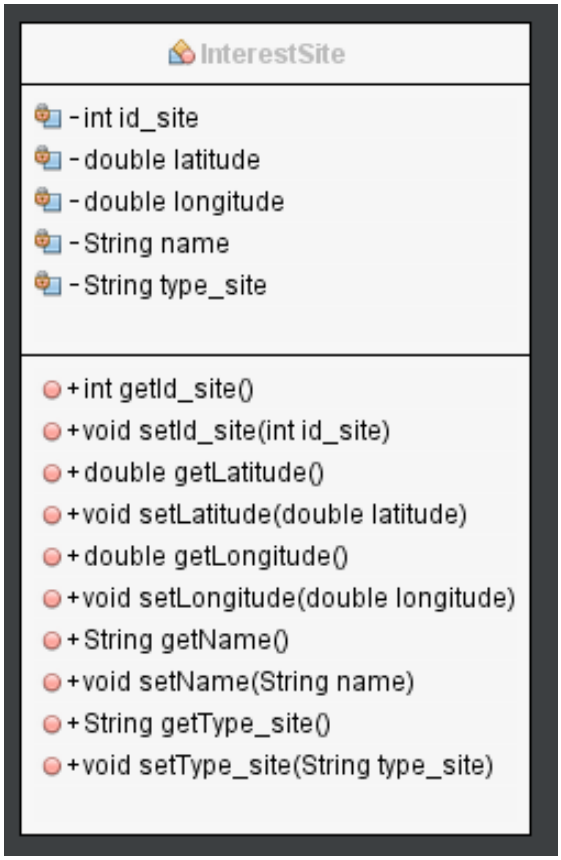
A continuación se describen los paquetes encargados de mapear el modelo de la base de datos.

Figura 26. Clases del paquete models de SITApp



Fuente: Esta investigación.

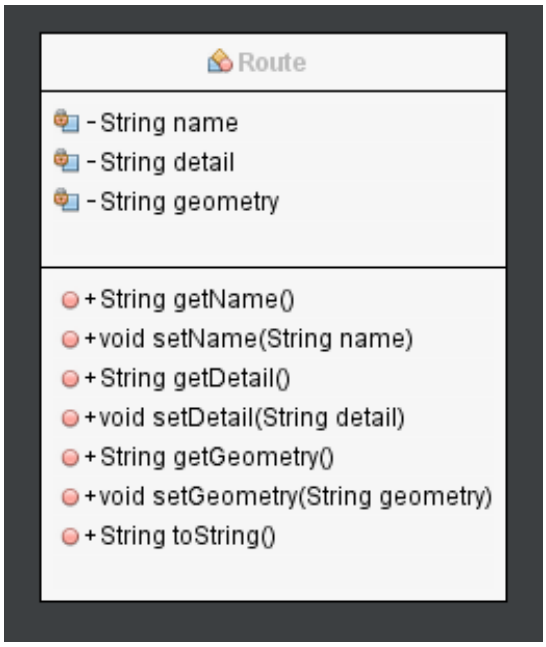
Figura 27. Clase InterestSite.java



Fuente: Esta investigación.

Clase que representa la tabla InterestSites de la base de datos.

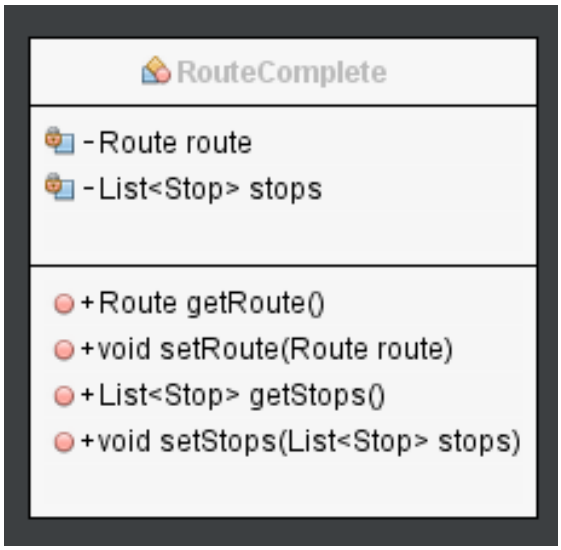
Figura 28. Clase Route.java



Fuente: Esta investigación.

Clase que representa la tabla Routes de la base de datos.

Figura 29. Clase RouteComplete.java



Fuente: Esta investigación.

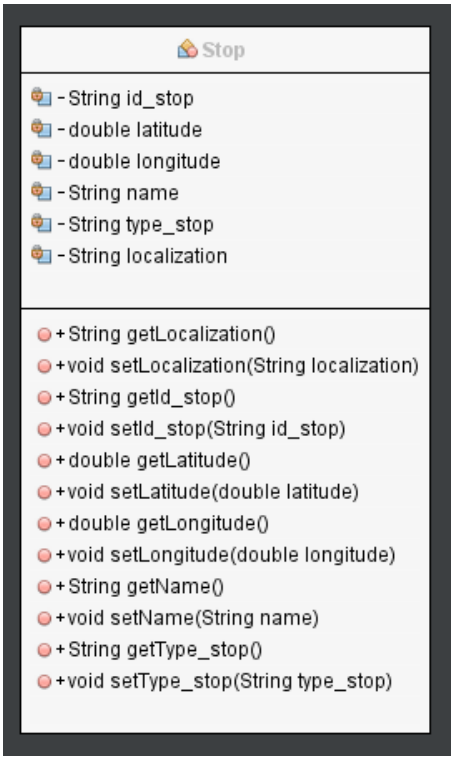
Clase que representa la ruta con todos sus paraderos.

Figura 30. Clase RouteStops.java

Fuente: Esta investigación.

Clase que representa la ruta los paraderos de origen y destino.

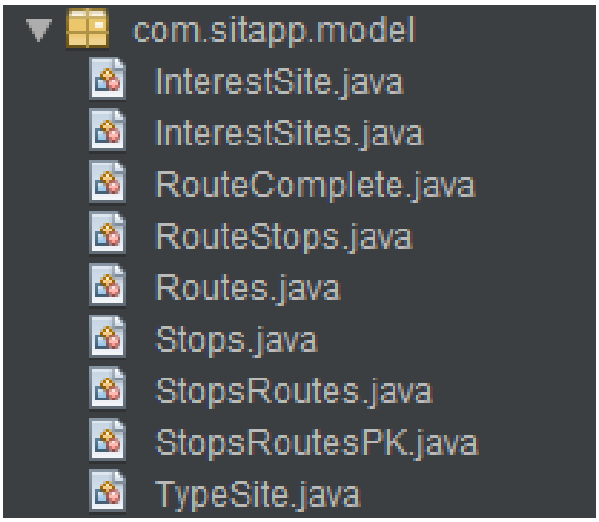
Figura 31. Clase Stop.java



Fuente: Esta investigación.

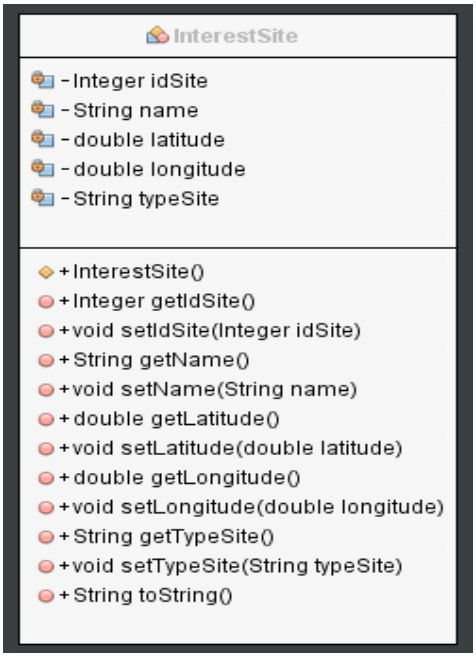
Clase que representa la tabla Stops de la base de datos.

Figura 32. Clases del paquete model del SITApp Web Service



Fuente: Esta investigación.

Figura 33. Clase InterestSite.java



Fuente: Esta Investigación.

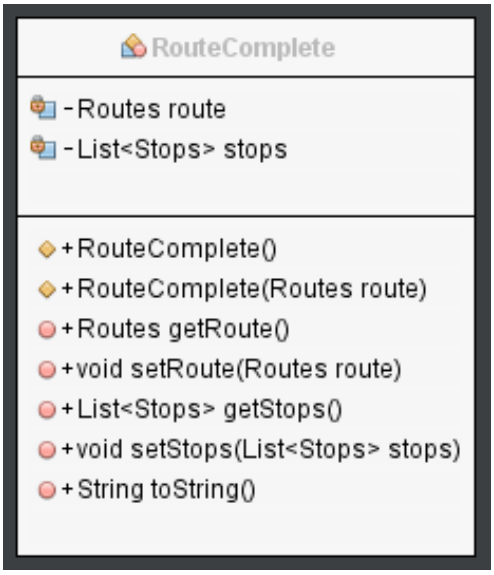
Esta clase es una adaptación de la clase InterestSites, fue desarrollada con el objetivo de enviar la menor cantidad de datos en el archivo JSON hacia SITApp.

Figura 34. Clase InterestSites.java

Fuente: Esta investigación.

Clase que representa la tabla InterestSites de la base de datos de SITApp.

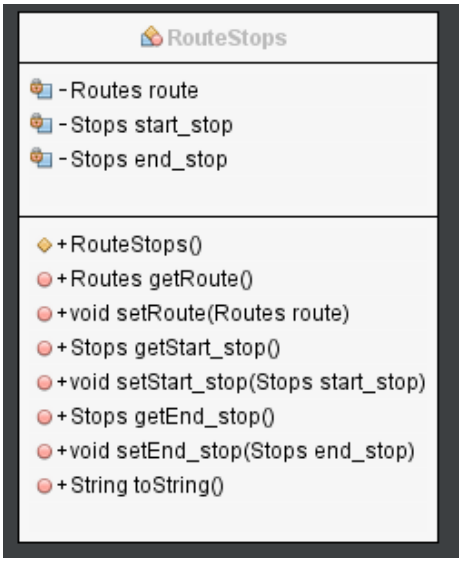
Figura 35. Clase RouteComplete.java



Fuente: Esta investigación.

Clase que representa una ruta con todos sus paraderos asociados, fue desarrollada con el objetivo de enviar la menor cantidad de datos en el archivo JSON hacia SITApp.

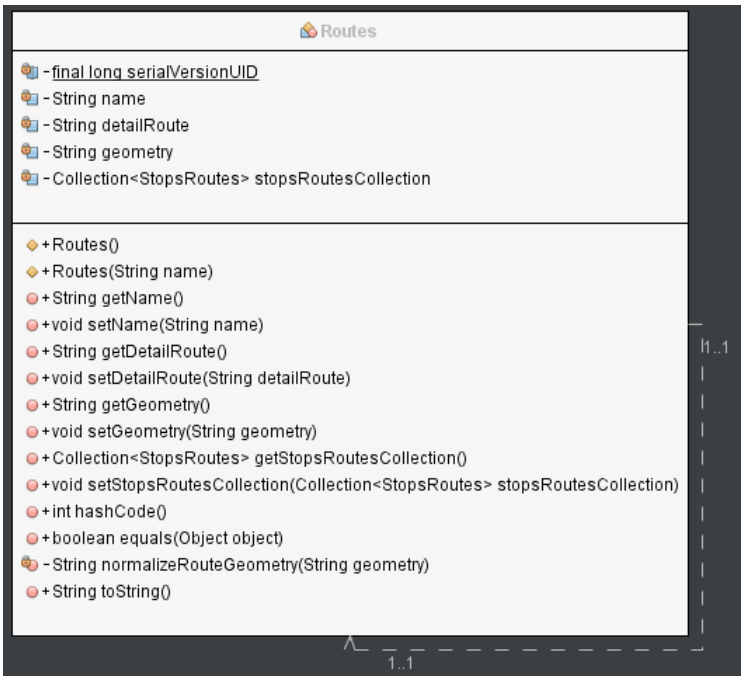
Figura 36. Clase RouteStops.java



Fuente: Esta investigación.

Clase que representa a la ruta con el paradero de origen y destino, fue desarrollada con el objetivo de enviar la menor cantidad de datos en el archivo JSON hacia SITApp.

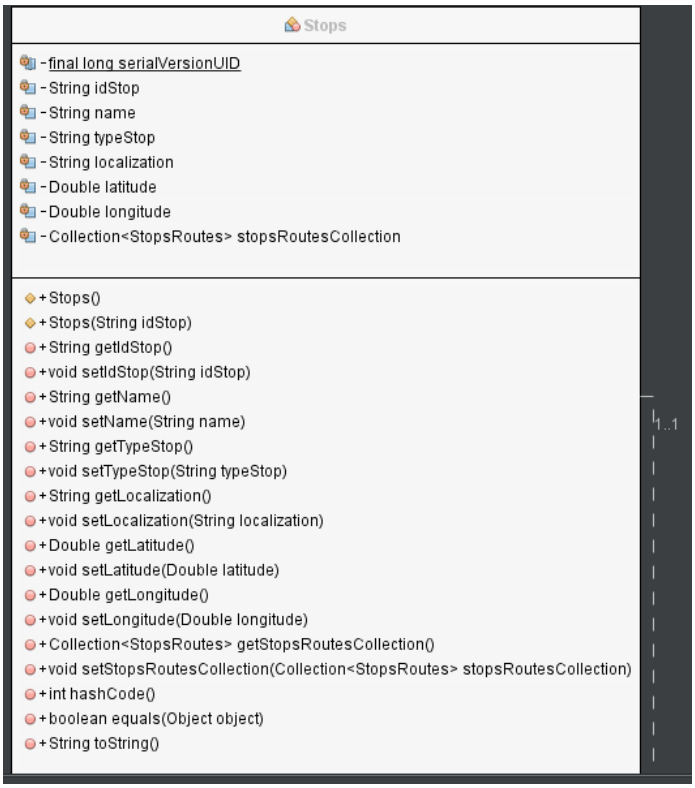
Figura 37. Clase Routes.java



Fuente: Esta investigación.

Clase que representa a la tabla Routes de la base de datos de SITApp.

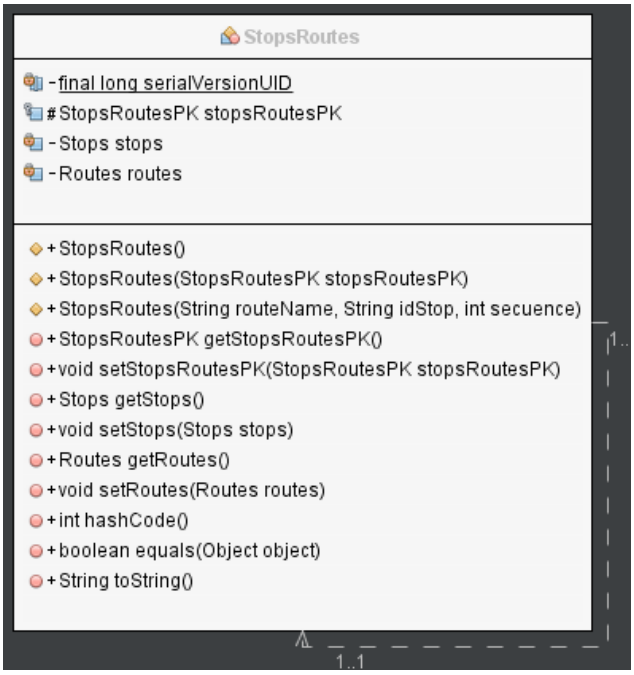
Figura 38. Clase Stops.java



Fuente: Esta investigación.

Clase que representa la tabla Stops de la base de datos de SITApp.

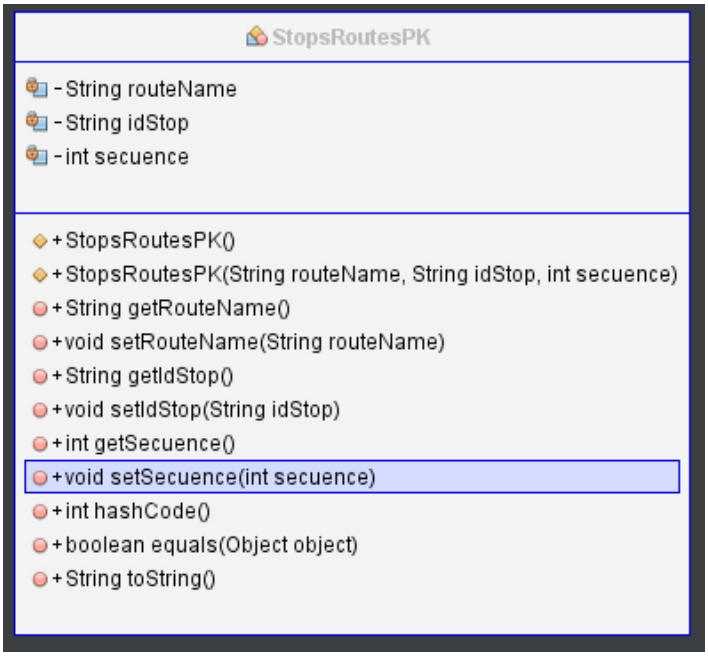
Figura 39. Clase RouteStops.java



Fuente: Esta investigación.

Clase que representa la tabla intermedia que conecta las rutas con cada uno de sus paraderos llamada RouteStops en la base de datos de SITApp.

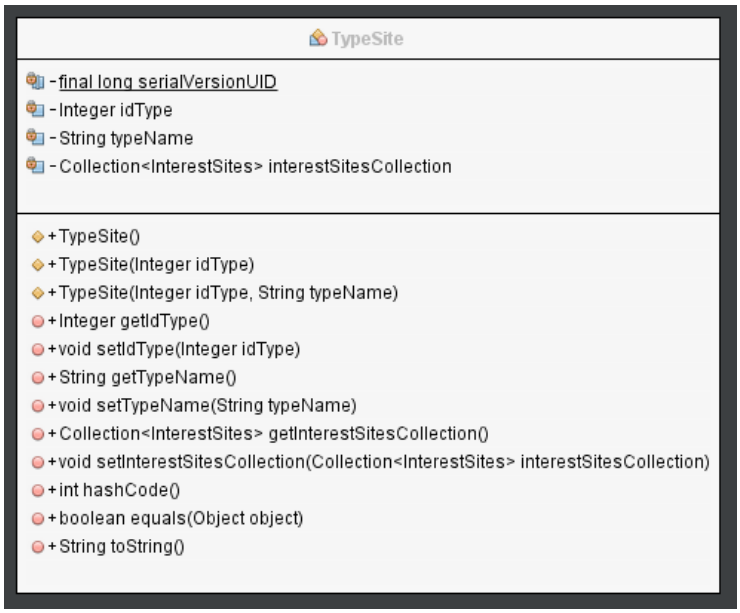
Figura 40. Clase RouteStopsPK.java



Fuente: Esta investigación.

Clase creada por Java EE para representar la llave primaria de la clase intermedia StopsRoutes de la base de datos de SITApp.

Figura 41. Clase TypeSite.java



Fuente: Esta investigación.

Clase que representa la tabla TypeSites de la base de datos de SITApp.

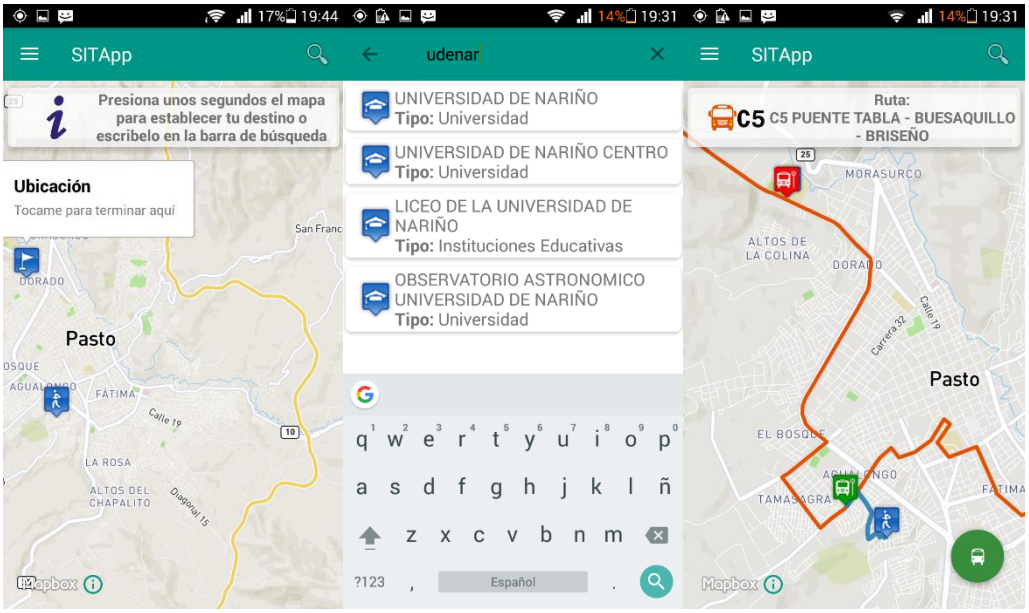
2. FUNCIONALIDADES SITAPP ANDROID

A continuación se describen las funcionalidades de SITApp Android:

2.1. Búsqueda por un sitio de interés

Esta funcionalidad permite al usuario realizar la búsqueda de rutas mediante la posición GPS del dispositivo y un sitio de interés en la ciudad, este sitio puede ser buscando usando el buscador semántico o presionando prolongadamente el mapa para establecerlo.

Figura 42. Ejemplo búsqueda mediante GPS y un sitio de interés.



Fuente: Esta investigación.

El ciclo de vida normal de esta funcionalidad pasa principalmente por la clase **MainActivity.java** que se encuentra en el paquete **com.udenar.sitapp.views** descrito anteriormente.

Cuando el usuario usa el buscador semántico la aplicación usa el escuchador de eventos del componente SearchView como se muestra a continuación:

```
searchView.setOnQueryTextListener(new
SearchView.OnQueryTextListener() {
    // When user sends query
    @Override
    public boolean onQueryTextSubmit(String query) {
        // Código omitido
        launchSitesService(query);
        return true;
```

Este código llama al método **lauchSitesService()** que se encarga de establecer comunicación con el WebService utilizando la librería retrofit como se muestra a continuación:

```
public void launchSitesService(String query) {
    // Build Retrofit to support web service communication
    final Retrofit retrofit = new Retrofit.Builder()
            .baseUrl(web service)
            .addConverterFactory(GsonConverterFactory.create())
            .build();
    // Create InterestSite service
    InterestSitesService service =
retrofit.create(InterestSitesService.class);
    // Send request to SITApp Web Service
    Call<List<InterestSite>> call = service.getSites(query);
    // Bring results from request
    call.enqueue(new Callback<List<InterestSite>>() {
        @Override
        public void onResponse(Call<List<InterestSite>> call,
                               Response<List<InterestSite>>
response) {
            // Get InterestSites list
            List<InterestSite> sites = response.body();
            // If there's results
            if (sites.size() > 0) {
                // Closing keyboard
                activity.closeKeyboard();
                // Get Fragment Manager
                final FragmentTransaction transaction =

getSupportFragmentManager().beginTransaction();

                // Create Sites Fragment
                sitesFragment = new InterestSitesFragment();
                sitesFragment.setSites(sites);
                sitesFragment.setActivity(activity);

                // Add Sites Fragment to layout container
                transaction.replace(R.id.container, sitesFragment,
TAG_SITE_FRAGMENT);
                transaction.commit();
            }
            // If there's no results
            else {
                Toast.makeText(activity,
getText(R.string.no results), Toast.LENGTH LONG).show();
            }
        }
}
```

Si hay respuesta por parte del Web Service esto significa que encontró uno o más sitios de interés, por lo tanto haciendo uso de fragments, reemplaza el mapa por el listado de sitios de interés encontrados, si el usuario selecciona uno de estos sitios como su lugar de destino se dispara el método **onClick()** de la clase **InterestSitesHolder.java,** a continuación el fragmento de código del método para esta opción de búsqueda:

```
if (context.getState() == 0) {
    if (context.getMyLocation().getPosition() != null) {
        context.getStart point()
                .title(context.getMyLocation().getTitle())
                .position(context.getMyLocation().getPosition())
                .icon(context.getMyLocation().getIcon());
        context.getEnd point().title(site.getName())
                .position(new LatLng(site.getLatitude(),
                        site.getLongitude()))
                .snippet("Tipo: : " + site.getType_site())
                .icon(icon);
        context.setLast state(0);
        context.launchRouteService();
    } else {
        // ALERTA DE ERROR CÓDIGO OMITIDO
    }
}
```

El código establece el origen y destino para la búsqueda de rutas y posteriormente llama al método **lauchRouteService()** que se encuentra en la clase **MainActivity.java.** Este método realiza la petición al Web Service y si hay una respuesta significa que encontró una o más rutas, por lo tanto haciendo uso de fragments muestra el mapa con la ruta creada, a continuación el fragmento de código que realiza esta operación:

```
public void launchRouteService() {
    reloadMapBoxFragment();
    final Retrofit retrofit = new Retrofit.Builder()
            .baseUrl(web service)
            .addConverterFactory(GsonConverterFactory.create())
            .build();
    RouteService service = retrofit.create(RouteService.class);
    Call<List<RouteStops>> call =
service.getRoute(start_point.getPosition().getLatitude(),
            start_point.getPosition().getLongitude(),
            end point.getPosition().getLatitude(),
            end point.getPosition().getLongitude());
    //Toast.makeText(activity, getString(R.string.search route),
Toast.LENGTH SHORT).show();
    call.enqueue(new Callback<List<RouteStops>>() {
        @Override
        public void onResponse(Call<List<RouteStops>> call,
Response<List<RouteStops>> response) {
            // Get result
            routes = response.body();
            // If there aren't Routes
            if (routes == null) {
                Toast.makeText(activity, R.string.no route found,
Toast.LENGTH LONG).show();
                setState(0);
                setLast_state(0);
                searchItem.collapseActionView();
```

```
            }
            // If some Route was found
            else {
                final route = routes.get(0);
                createRoute(final_route);
                if (map != null) {
                    map.clear();
                    map.addMarker(start point);
                    map.addMarker(end point);
                    map.addMarker(start stop);
                    map.addMarker(end stop);
                    map.addPolyline(route);
                    activity.drawDirections(
                            start point.getPosition(),
                            start_stop.getPosition());
                    CameraPosition position = new
CameraPosition.Builder()
                            .target(start point.getPosition())
                            .zoom(13)
                            .build();
                    map.animateCamera(CameraUpdateFactory
                            .newCameraPosition(position), 10000);
}
//CÓDIGO OMITIDO
```

El método **createRoute()** pertenece a la clase **MainActivity.java** y se encarga de tomar los atributos del objeto **routeStops** los cuales son la geometría del recorrido de la ruta y los dos paraderos, paradero inicial y final, e instanciarlos como una poli-línea y dos marcadores para posteriormente graficarlos en el mapa, a continuación se muestra el código de este método:

```
public void createRoute(RouteStops routeStops) {
    activity.cardView.setVisibility(View.VISIBLE);

activity.setCardViewResources(routeStops.getRoute().getDetail());
    // Create an Icon object for the marker to use
    IconFactory iconFactory = IconFactory.getInstance(activity);
    Icon icon = iconFactory.fromResource(R.drawable.busstopi);
    // Set attributes to start_stop from routeStops
    start_stop.title(routeStops.getStart_stop().getName())
            .snippet(routeStops.getStart_stop().getLocalization())
            .position(new
LatLng(routeStops.getStart stop().getLatitude(),
                    routeStops.getStart stop().getLongitude()))
            .icon(icon);
    icon = iconFactory.fromResource(R.drawable.busstopf);
    // Set attributes to end stop from routeStops
    end_stop.title(routeStops.getEnd_stop().getName())
            .snippet(routeStops.getEnd_stop().getLocalization())
            .position(new
LatLng(routeStops.getEnd stop().getLatitude(),
                    routeStops.getEnd stop().getLongitude()))
            .icon(icon);
    route = createRoutePolyline(routeStops.getRoute());
}
```

Al recargarse el fragment que contiene al mapa, este hace que se active el método **onMapReady()** que se encuentra en la clase **MainActivity.java**. Este método es el encargado de graficar los marcadores y las poli-líneas en el mapa y además grafica las indicaciones que el usuario debe seguir para llegar desde su posición GPS hasta el primer paradero, a continuación el fragmento de código que realiza esta operación:

```
// If some route was found then we paint all markers and polylines
in map
if (activity.getState() == 4
        && start_point.getPosition() != null
        && end_stop.getPosition() != null
        && start_stop.getPosition() != null
        && end_point.getPosition() != null) {
    map.clear();
    map.addMarker(start_point);
    map.addMarker(end_point);
    map.addMarker(start_stop);
    map.addMarker(end_stop);
    map.addPolyline(route);
    //Draw directions
    if (last_state == 0) {

activity.drawDirections(start_point.getMarker().getPosition(),
                start_stop.getMarker().getPosition());
    } else {

activity.drawDirections(start_point.getMarker().getPosition(),
                start_stop.getMarker().getPosition());
        activity.drawDirections(end_stop.getMarker().getPosition(),
                end_point.getMarker().getPosition());
    }
    CameraPosition position = new CameraPosition.Builder()
            .target(start_stop.getPosition())
            .zoom(13)
            .build();
    map.setCameraPosition(position);
    position = new CameraPosition.Builder()
            .target(start_point.getPosition())
            .zoom(13)
            .build();
    map.animateCamera(CameraUpdateFactory
            .newCameraPosition(position), 10000);
}
```

Si el usuario opta por establecer su destino presionando prolongadamente el mapa se activa el método **onMapLongClick()** que se encuentra en la clase **MainActivity.java**, a continuación un fragmento del código de este método:

```
map.setOnMapLongClickListener(new
MapboxMap.OnMapLongClickListener() {
    @Override
    public void onMapLongClick(@NonNull LatLng point) {
        if (activity.getState() == 0) {
            // CÓDIGO OMITIDO
```

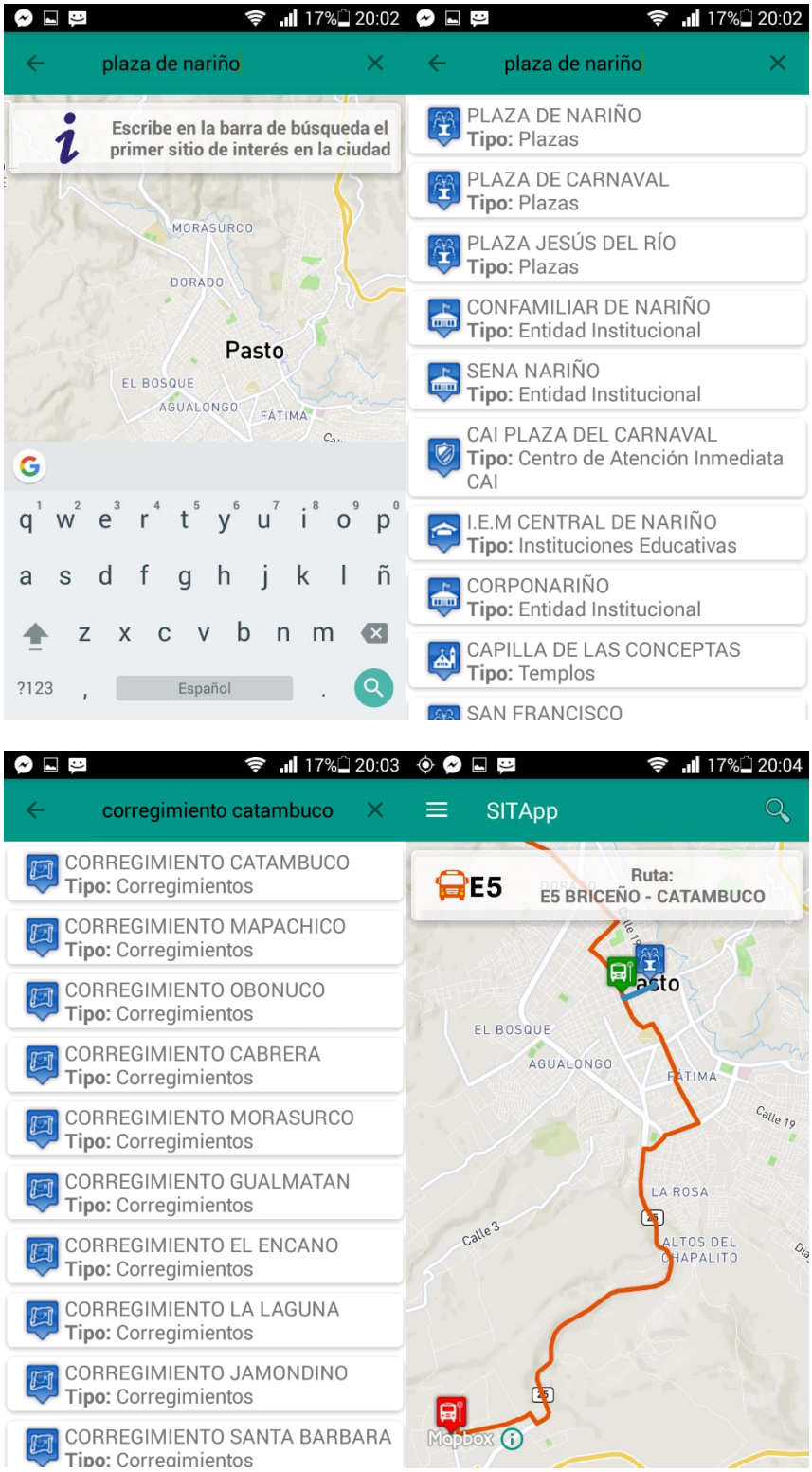
```
                end point.position(point);
                end point.title(getString(R.string.point));
                end point.snippet(getString(R.string.end travel));
                IconFactory iconFactory =
IconFactory.getInstance(activity);
                Icon icon =
iconFactory.fromResource(R.drawable.site);
                end point.getMarker().setIcon(icon);
                map.addMarker(end point);
                map.selectMarker(end point.getMarker());
                map.setOnInfoWindowClickListener(
                        new MapboxMap.OnInfoWindowClickListener() {
                            @Override
                            public boolean
onInfoWindowClick(@NonNull Marker marker) {
                                if
(map.getSelectedMarkers().contains(end point.getMarker())) {
                                    end point.title("Destino");
                                activity.launchRouteService();
                                    return true;
                                } else {
                                    return false;
                                }
                            }});}
```

Una vez se presiona la etiqueta que aparece encima del marcador en el mapa se establece el destino y se llama al método **launchRouteService()** de la clase **MainActivity.java** que ya fue descrito anteriormente.

2.2. Búsqueda por dos sitios de interés

Esta funcionalidad permite al usuario realizar una búsqueda de dos sitios de interés haciendo uso del buscador semántico para establecer el punto de origen y el punto de destino, en la Figura 43 se puede observar un ejemplo de esta funcionalidad:

Figura 43. Ejemplo búsqueda mediante dos sitios de interés



Fuente: Esta investigación.

Al igual que la función anterior se utiliza el buscador semántico para realizar la búsqueda de los dos sitios, pero cuando el usuario selecciona el sitio de interés del listado recibido, se dispara el siguiente fragmento de código del método **onClick()** de la clase **InterestSiteHolder()**:

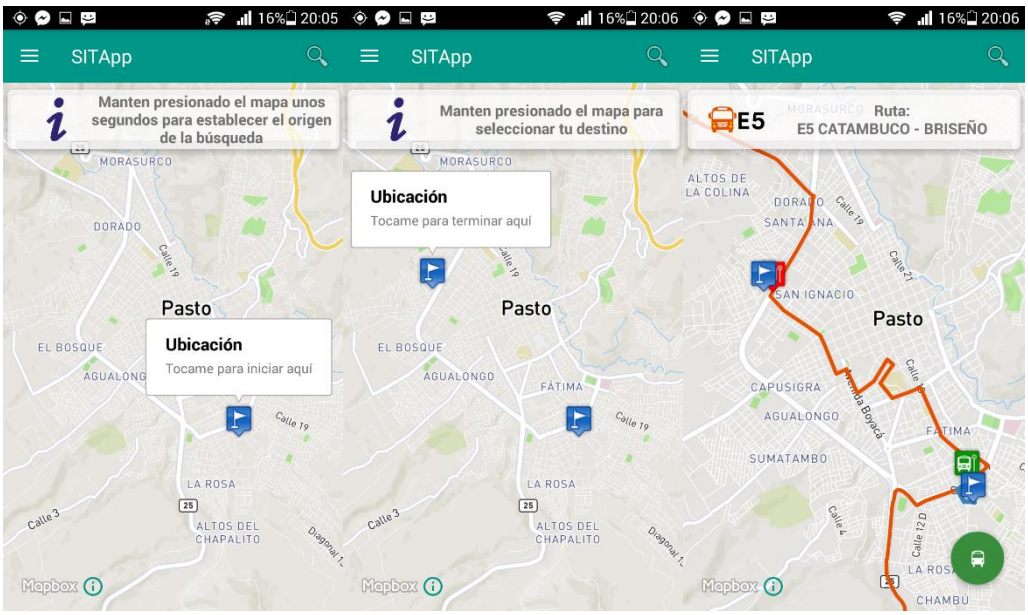
```
// If search by two sites is enabled, for first site
else if (context.getState() == 1) {
    context.getStart point().title(site.getName())
            .position(new LatLng(site.getLatitude(),
                    site.getLongitude()))
            .snippet("Tipo: " + site.getType_site())
            .icon(icon);
    context.setState(2);
    context.setLast state(1);
    context.txtCard.setText("Escribe en la barra de búsqueda el
segundo sitio de interés en la ciudad");
    context.getSearchView().setQuery("", false);
    context.cardView.setVisibility(View.VISIBLE);
    /*InterestSitesAdapter adapter = new
InterestSitesAdapter(context,
            R.layout.list_item_sites, new
ArrayList<InterestSite>());
    context.getSitesFragment().setAdapter(adapter);*/
    context.reloadMapBoxFragment();
}
// If search by touch two sites is enabled, for second site
else if (context.getState() == 2) {
    context.getEnd point().title(site.getName())
            .position(new LatLng(site.getLatitude(),
                    site.getLongitude()))
            .snippet("Tipo: " + site.getType site())
            .icon(icon);
    context.setLast state(2);
    context.launchRouteService();
}
```

Cuando el estado de la aplicación es 1, esto significa que está buscando el primer sitio de interés por lo tanto al ser seleccionado este primer sitio se cambia el estado a 2, se establece el punto de origen de la búsqueda y se recarga el mapa para mostrarle al usuario gráficamente cual fue su selección, ahora el usuario debe buscar el segundo sitio, vuelve a realizarse el proceso de búsqueda y cuando el segundo sitio es seleccionado se dispara la segunda parte del método cuando el estado es 2, se establece el punto de destino de la búsqueda y se llama al método **launchRouteService()** de la clase **MainActivity.java** que se encarga de comunicarse con el Web Service para buscar si hay una o más rutas que lleven al usuario desde el origen hasta el destino establecido. Este proceso de búsqueda de rutas se realiza de la misma manera que la funcionalidad anterior.

2.3. Búsqueda por tocar dos puntos

Esta funcionalidad permite al usuario establecer su origen y su destino en la ciudad presionando prolongadamente el mapa en la aplicación, en la Figura 44 se puede observar un ejemplo de la funcionalidad:

Figura 44. Ejemplo de búsqueda tocando dos puntos en el mapa



Fuente: Esta investigación.

A continuación un fragmento del método **setOnMapLongClickListener()** que se encarga de gestionar las operaciones cuando el mapa es presionado prolongadamente, esta función se encuentra en la clase **MainActivity.java**.

```
if (activity.getState() == 3) {
    if (number markers == 2) {
        //map.clear();
        map.deselectMarkers();
        if (start_point.getPosition() != null) {
            if (map.getMarkers().contains(start_point.getMarker()))
{
                map.setOnInfoWindowClickListener(null);
                map.removeMarker(start point.getMarker());
            }
        }
        start point.position(point);
        start point.title(getString(R.string.point));
        start_point.snippet(getString(R.string.start_travel));
        start_point.getMarker().setFlat(true);
        IconFactory iconFactory =
IconFactory.getInstance(activity);
        Icon icon = iconFactory.fromResource(R.drawable.site);
        start point.getMarker().setIcon(icon);
        map.selectMarker(start point.getMarker());
        map.addMarker(start_point);
```

```
        map.setOnInfoWindowClickListener(
                new MapboxMap.OnInfoWindowClickListener() {
                    @Override
                    public boolean onInfoWindowClick(@NonNull
Marker marker) {
                        start_point.snippet("Lat: "
                                +
start_point.getPosition().getLatitude()
                                + "\nLong: "
                                +
start_point.getPosition().getLongitude());
                        number_markers--;
                        map.setOnInfoWindowClickListener(null);
                        map.deselectMarkers();
                        activity.txtCard.setText("Manten presionado
el mapa para seleccionar tu destino");
                        return true;
                    }
                });
    } else if (number_markers == 1) {
        map.deselectMarkers();
        //map.clear();
        if (end_point.getPosition() != null) {
            if (map.getMarkers().contains(end_point.getMarker())) {
                map.removeMarker(end_point.getMarker());
            }
        }
        end_point.position(point);
        end_point.title(getString(R.string.point));
        end_point.snippet(getString(R.string.end_travel));
        IconFactory iconFactory =
IconFactory.getInstance(activity);
        Icon icon = iconFactory.fromResource(R.drawable.site);
        end_point.getMarker().setIcon(icon);
        map.addMarker(end_point);
        map.selectMarker(end_point.getMarker());
        map.setOnInfoWindowClickListener(
                new MapboxMap.OnInfoWindowClickListener() {
                    @Override
                    public boolean onInfoWindowClick(@NonNull
Marker marker) {
                        if
(map.getSelectedMarkers().contains(end_point.getMarker())) {
                            end_point.title("Destino");
                            end_point.snippet("Lat: "
                                    +
end_point.getPosition().getLatitude()
                                    + "\nLong: "
                                    +
end_point.getPosition().getLongitude());
                            number_markers--;
                            activity.launchRouteService();
                            activity.setLast_state(3);
                            map.setOnInfoWindowClickListener(null);
                            return true;
                        } else {
                            return false;
```

El método **onInfoWindowsClick()** es llamado cuando el usuario presiona el título del marcador para establecer el origen inicialmente, este marcador se fija y el usuario tiene la posibilidad de presionar nuevamente la pantalla prolongadamente para establecer su destino de la misma manera que el origen. Una vez fijado el destino se llama a la función **launchRouteService()** de la clase **MainActivity.java** que se encarga de comunicarse con el Web Service para buscar si hay una o más rutas que lleven al usuario desde el origen hasta el destino establecido, que funciona de la misma manera que en las dos opciones anteriores de búsqueda.

2.4. Mostrar recorridos de rutas

Esta funcionalidad permite al usuario saber los recorridos de las rutas y todos sus paraderos autorizados. La búsqueda está discriminada por rutas estratégicas o rutas complementarias, en la Figura 45 se puede observar un ejemplo de esta funcionalidad.

Figura 45. Ruta C9, recorrido y parada

Fuente: Esta investigación.

A continuación un fragmento del método **getStrategicsComplementariesRoutes()** encargado de gestionar la peticion hacia el Web Service de SITApp recibiendo como respuesta un listado de rutas con sus diferentes paraderos.

```
private void getStrategicsComplementariesRoutes(String search) {

    final Retrofit retrofit = new Retrofit.Builder()
            .baseUrl(web service)
            .addConverterFactory(GsonConverterFactory.create())
            .build();
    TypeRouteService service =
retrofit.create(TypeRouteService.class);
```

```
    Call<List<RouteComplete>> call = service.getRoutes(search);
    //Toast.makeText(activity, getString(R.string.search route),
Toast.LENGTH SHORT).show();

    call.enqueue(new Callback<List<RouteComplete>>() {
        @Override
        public void onResponse(Call<List<RouteComplete>> call,
Response<List<RouteComplete>> response) {
            List<RouteComplete> routes = response.body();
            if (!routes.isEmpty()) {
                final FragmentTransaction transaction =

getSupportFragmentManager().beginTransaction();

                // Create Routes Complete Fragment
                routesCompleteFragment = new
RoutesCompleteFragment();
                routesCompleteFragment.setRoutes(routes);
                routesCompleteFragment.setActivity(activity);

                // Add Sites Fragment to layout container
                transaction.replace(R.id.container,
routesCompleteFragment, TAG_ROUTES_COMPLETE_FRAGMENT);
                transaction.commit();
            }
        }

        @Override
        public void onFailure(Call<List<RouteComplete>> call,
Throwable t) {

        }
    });
}
```

Se crea y se ubica como fragment principal al fragment **routesCompleteFragment** que se encarga de mostrar el listado de rutas al usuario, cuando es seleccionada una de las rutas es llamado el método **onClick()** de la clase **RoutesCompleteHolder** que se muestra a continuación:

```
public void onClick(View view) {

    // 5. Handle the onClick event for the ViewHolder
    if (this.routeStops != null) {
        IconFactory iconFactory = IconFactory.getInstance(context);
        Icon icon = iconFactory.fromResource(R.drawable.busstop);
        context.getRoute_stops().clear();
        for (int i = 1; i < routeStops.getStops().size(); i++) {
            if (i < routeStops.getStops().size() - 1) {
                context.getRoute stops().add(
                        new MarkerViewOptions()

.title(routeStops.getStops().get(i).getName())
                                .position(new
```

```
LatLng(routeStops.getStops().get(i).getLatitude(),

routeStops.getStops().get(i).getLongitude()))

.snippet(routeStops.getStops().get(i).getLocalization())
                                    .icon(icon)
                );
            } else {
                icon =
iconFactory.fromResource(R.drawable.busstopf);
                context.getRoute_stops().add(
                        new MarkerViewOptions()

.title(routeStops.getStops().get(i).getName())
                                    .position(new
LatLng(routeStops.getStops().get(i).getLatitude(),

routeStops.getStops().get(i).getLongitude()))

.snippet(routeStops.getStops().get(i).getLocalization())
                                    .icon(icon)
                );
            }
        }
        icon = iconFactory.fromResource(R.drawable.busstopi);
        context.getRoute_stops().add(
                new MarkerViewOptions()

.title(routeStops.getStops().get(0).getName())
                        .position(new
LatLng(routeStops.getStops().get(0).getLatitude(),

routeStops.getStops().get(0).getLongitude()))

.snippet(routeStops.getStops().get(0).getLocalization())
                        .icon(icon)
        );
        context.setLast_state(5);

context.setRoute(context.createRoutePolyline(routeStops.getRoute())
);

context.setCardViewResources(routeStops.getRoute().getDetail());
        context.reloadMapBoxFragment();
        context.cardView.setVisibility(View.VISIBLE);
    }
}
```

Al final de este método se recarga el fragment de Mapbox e inmediatamente se ejecuta el método asíncrono **onMapReady()**, específicamente se ejecuta el siguiente fragmento de código encargado de mostrar en el mapa el recorrido de la ruta y cada uno de los paraderos autorizados.

```
    else if ((activity.getState() == 5 || activity.getState() == 6)
            && activity.getLast state() == 5) {
        map.clear();
        for (MarkerViewOptions marker : activity.getRoute_stops())
{
            map.addMarker(marker);
        }
        map.addPolyline(route);
        CameraPosition position = new CameraPosition.Builder()
                .target(new LatLng(1.2084236, -77.2787569))
                .zoom(12)
                .build();
        map.setCameraPosition(position);
    }
}
```

3. ALGORITMOS DE SITAPP WEB SERVICE

SITApp Web Service alberga los algoritmos de conexión y consulta hacia la base de datos, la ontología, además los algoritmos de inferencia y de cruce de rutas.

3.1. Algoritmo de conexión y búsqueda de sitios en ontología

La clase **ConectionOntology.java** es la encargada de gestionar las conexiones y las consultas hacia la ontología, a continuación el código fuente de esta clase.

```
public class ConectionOntology {

    Query query;
    QueryExecution qexec;
    OntModel model;

    public OntModel getModel() {
        return model;
    }

    public void setModel(OntModel model) {
        this.model = model;
    }

    public OntModel start() {
        model =
ModelFactory.createOntologyModel(OntModelSpec.OWL_MEM);
        InputStream in =
FileManager.get().open("/home/user/sitapp/sitapp.owl");
        if (in == null) {
            throw new IllegalArgumentException("Archivo No
Encontrado");
```

```
        }
        model.read(in, "");
        return model;
    }

    public ResultSet consult(String consulta) {
        query = QueryFactory.create(consulta);
        qexec = QueryExecutionFactory.create(query, model);
        ResultSet results = qexec.execSelect();
        return results;
    }

    public void stop() {
        qexec.close();
    }
}
```

La clase **OntologyQueries.java** posee el algoritmo de búsqueda e inferencia en SPARQL.

```
public class OntologyQueries {

    private ConectionOntology ontologia;

    public OntologyQueries(ConectionOntology ontologia) {
        this.ontologia = ontologia;
    }

    public ConectionOntology getConectionOntology() {
        return ontologia;
    }

    public void setConectionOntology(ConectionOntology ontologia)
{
        this.ontologia = ontologia;
    }


    public List<String> searchSites(String query){
        List<String> sites = new ArrayList<>();
        String[] words = query.split(" ");
        String filter = "";
        for (int i = 0; i < words.length ; i++){
            if (i != words.length-1){
                filter += "REGEX (?sin, \"" + words[i] + "\")||"
                        + "REGEX (?pal, \"" + words[i] + "\")||";
```

```
            } else {
                filter += "REGEX (?sin, \"" + words[i] + "\")||"
                        + "REGEX (?pal, \"" + words[i] + "\")";

            }
        }
        String sparql = "PREFIX
ont:<http://www.semanticweb.org/sitapp.owl#>\n"
                + "SELECT distinct ?id\n"
                + "where\n"
                + "{\n"
                + "?Sitios_interes ont:id ?id.\n"
                + "?Sitios_interes ont:sinonimos ?sin.\n"
                + "?Sitios_interes ont:es_descrito_por
?keyword.\n"
                + "?keyword ont:palabra ?pal.\n"
                + "FILTER (\n"
                + filter
                + ").\n"
                + "}\n"
                + "group by ?id ?sin ?keyword ?pal\n"
                + "order by ?id";
        ResultSet rs = ontologia.consult(sparql);
        while (rs.hasNext()){
            QuerySolution sol = rs.nextSolution();
            sites.add(sol.get("?id").toString().

replace("^^http://www.w3.org/2001/XMLSchema#string", ""));
        }
        return sites;
    }
}
```

La consulta SPARQL relaciona las entidades Sitios Interés y Palabras Clave mediante las relaciones definidas en el modelo, internamente la ontología busca todas las coincidencias en la consulta y retorna como resultado un listado de sitios de interés según los criterios de búsqueda.

3.2. Servicio de búsqueda de sitios de interés

Este servicio web se encuentra en la clase **InterestSitesFacadeREST.java**, encargado de recibir la consulta enviada desde SITApp Android y llamar a la case **OntologyQueries.java** para que se realice la inferencia en la ontología y posteriormente se consulta a la base de datos el listado de sitios de interés retornado por la ontología con el fin de traer todos los

datos georreferenciados almacenados referente a cada uno de los sitios de interés y se devuelve este listado de sitios mediante un archivo tipo JSON.

```
@GET
@Path("sites/{query}")
@Produces(MediaType.APPLICATION_JSON + ";charset=utf-8")
public List<InterestSite> find(@PathParam("query") String query) {
	System.out.println("Web Service Llamado, Consulta: " +
query);
	ConectionOntology ont = new ConectionOntology();
	ont.start();
	OntologyQueries querys = new OntologyQueries(ont);
	String query2 = query;
	query = deleteDeadWords(query);
	List<String> sitesIds =
querys.searchSites(query.toUpperCase());
	ont.stop();
	Query nativeQuery;
	if (sitesIds.isEmpty()) {
		nativeQuery = em.createNativeQuery("SELECT id_site,
name, type_site, latitude, "
				+ "longitude, jarowinkler('"
				+ query2
				+ "', name ) FROM interest_sites "
				+ "order by 6 desc "
				+ "limit 10");
	} else {
		String cad = "";
		for (int i = 0; i < sitesIds.size(); i++) {
			if (i != sitesIds.size() - 1) {
				cad += sitesIds.get(i) + ",";
			} else {
				cad += sitesIds.get(i);
			}
		}
		nativeQuery = em.createNativeQuery("SELECT id_site,
name, type_site"
				+ ", latitude, longitude, jarowinkler('"
				+ query2
				+ "', name ) FROM interest_sites "
				+ "WHERE id_site in ("
				+ cad
				+ ") order by 6 desc "
				+ "limit 30;");
	}
	List<Objects[]> sitesInit = nativeQuery.getResultList();
```

```
      List<InterestSite> sites = new ArrayList();
      for (Object[] o : sitesInit) {
            System.out.print(o[0]);
            InterestSites s =
(InterestSites)em.createNamedQuery("InterestSites.findByIdSite")
                        .setParameter("idSite", o[0])
                        .getSingleResult();
            InterestSite is = new InterestSite();
            is.setName(s.getName());
            is.setLatitude(s.getLatitude());
            is.setLongitude(s.getLongitude());
            is.setTypeSite(s.getTypeSite().getTypeName());
            sites.add(is);
      }
      return sites;
}
```

3.3. Servicio de búsqueda por tipo de ruta

Este servicio web gestiona la búsqueda en la base de datos la información referente a las rutas, estratégicas o complementarias, y los paraderos autorizados asociados a cada una de las rutas como se muestra en el siguiente fragmento de código de la clase **RoutesFacadeREST.java**.

```
@GET
@Path("byRouteType/{routeType}")
@Produces(MediaType.APPLICATION_JSON + ";charset=utf-8")
public List<RouteComplete> findByRouteType(@PathParam("routeType")
String routeType) {
      if (routeType.length() != 1) return null;
      Query qr;
      String sql = "SELECT name FROM routes WHERE name LIKE '" +
routeType.toUpperCase() + "%'";
      qr = em.createNativeQuery(sql);
      List<String> routes = qr.getResultList();
      if (routes.isEmpty()) {
            return null;
      }
      List<RouteComplete> finalList = new ArrayList<>();
      for (int i = 0; i < routes.size(); i++) {
            List<StopsRoutes> stopsroutes =
em.createNamedQuery("StopsRoutes.findByRouteName")
                        .setParameter("routeName", routes.get(i))
                        .getResultList();
            System.out.print(routes.get(i));
            if (!stopsroutes.isEmpty()) {
```

```
                RouteComplete custom = new
RouteComplete(stopsroutes.get(0).getRoutes());
                ArrayList<Stops> stops = new ArrayList<>();
                for (StopsRoutes s : stopsroutes) {
                     stops.add(s.getStops());
                }
                custom.setStops(stops);
                finalList.add(custom);
           }
     }

     return finalList;
}
```

3.4. Algoritmo de cruce de rutas

El algoritmo busca inicialmente 500 metros a la redonda para el punto de origen “A” y para el punto de destino “B” seleccionados por el usuario desde SITApp Android, luego analiza si alguna de las rutas posee la combinación de paraderos cercanos a los puntos “A” y “B” y retorna mediante un archivo JSON el listado de rutas que le sirven al usuario junto con dos paraderos para cada ruta, uno de origen y uno de destino.

```
@GET
@Path("route/{start_lat}/{start_long}/{end_lat}/{end_long}")
@Produces(MediaType.APPLICATION_JSON + ";charset=utf-8")
public List<RouteStops> routeAlgorithm(
          @PathParam("start_lat") Double start_lat,
          @PathParam("start_long") Double start_long,
          @PathParam("end_lat") Double end_lat,
          @PathParam("end_long") Double end_long) {

     // Building native query to find stops near Start Point 500
metters arround
     Query query = em.createNativeQuery("SELECT id_stop FROM stops
s\n"
               + "WHERE ST_CONTAINS(\n"
               + "ST_BUFFER(ST_POINT( ? , ? ), 0.005), \n"
               + "ST_POINT(s.latitude, s.longitude));");

     // Setting parameters
     query.setParameter(1, start_lat);
     query.setParameter(2, start_long);

     // Getting Result
     List<String> stops1 = query.getResultList();
```

```
	// If no results algorithm end
	if (stops1.isEmpty()) {
		return null;
	}

	// Making first condition 'stop1','stop2', ... ,'stopN'
	String first_condition = "";
	for (int i = 0; i < stops1.size(); i++) {
		if (i != stops1.size() - 1) {
			first_condition += "'" + stops1.get(i) + "',";
		} else {
			first_condition += "'" + stops1.get(i) + "'";
		}

	}

	System.out.println("Posicion Inicio: " + start_lat + "," +
start_long);
	System.out.println("Condicion #1: " + first_condition);

	// Building native query to find stops near End Point 500
metters arround
	query = em.createNativeQuery("SELECT id_stop FROM stops s\n"
			+ "WHERE ST_CONTAINS(\n"
			+ "ST_BUFFER(ST_POINT( ? , ? ), 0.005), \n"
			+ "ST_POINT(s.latitude, s.longitude));");

	// Setting parameters
	query.setParameter(1, end_lat);
	query.setParameter(2, end_long);

	// Getting Results
	List<String> stops2 = query.getResultList();

	// If no results algorithm end
	if (stops2.isEmpty()) {
		return null;
	}

	// Making second condition 'stop1','stop2', ... ,'stopN'
	String second_condition = "";
	for (int i = 0; i < stops2.size(); i++) {
		if (i != stops2.size() - 1) {
			second_condition += "'" + stops2.get(i) + "',";
		} else {
			second_condition += "'" + stops2.get(i) + "'";
```

```
            }
        }
        System.out.println("Posicion Llegada: " + end_lat + "," +
end_long);
        System.out.println("Condicion #2: " + second_condition);

        /**
         * Finding routes with two nearest stops, start stop and end
stop.
         * getting as result, its name, start stop id, end stop id,
start stop
         * sequence, end stop sequence, distance from start point to
start stop
         * and distance from end point to start stop
         */
        String qr = "SELECT\n"
                  + "t1.route_name, t1.id_stop, t2.id_stop,
t1.secuence,"
                  + "t2.secuence,"
                  + "ST_DISTANCE(ST_POINT( ? , ? ),"
                  + "ST_POINT(t1.latitude,t1.longitude)) + "
                  + "ST_DISTANCE(ST_POINT( ? , ? ),"
                  + "ST_POINT(t2.latitude,t2.longitude)) "
                  + "FROM\n"
                  + "(SELECT route_name, s.id_stop, latitude,
longitude, secuence "
                  + "FROM stops_routes sr JOIN stops s\n"
                  + "ON sr.id_stop=s.id_stop\n"
                  + "WHERE\n"
                  + "s.id_stop in\n"
                  + "( " + first_condition + " )"
                  + ")t1 JOIN\n"
                  + "(SELECT route_name, s.id_stop, latitude,
longitude, secuence "
                  + "FROM stops_routes sr JOIN stops s\n"
                  + "ON sr.id_stop=s.id_stop\n"
                  + "WHERE\n"
                  + "s.id_stop in "
                  + "( " + second_condition + " ))t2\n"
                  + "ON t1.route_name=t2.route_name AND t1.id_stop
!= t2.id_stop\n"
                  + "WHERE t1.secuence<t2.secuence\n"
                  + "ORDER BY 6;";
        System.out.print(qr);

        // Building native query
```

```
	query = em.createNativeQuery(qr);
	query.setParameter(1, start_lat);
	query.setParameter(2, start_long);
	query.setParameter(3, end_lat);
	query.setParameter(4, end_long);

	//List<Object[]> final_result = query.getResultList();
	// Getting result
	List<Object[]> result = query.getResultList();
	System.out.print(result.size());

	// If there are results
	if (!result.isEmpty()) {
		List<Object[]> oneResutPerRoute = new ArrayList<>();
		List<String> routesNames = new ArrayList<>();
		// Algorithm to get only a single result by route and
the best
		// of each one
		for (int i = 0; i < result.size(); i++) {
			if (i == 0) {
				oneResutPerRoute.add(result.get(i));
				routesNames.add(result.get(i)[0].toString());
			} else if
(!routesNames.contains(result.get(i)[0].toString())) {
				oneResutPerRoute.add(result.get(i));
				routesNames.add(result.get(i)[0].toString());
			}
		}

		List<RouteStops> routeFinal = new ArrayList<>();

		// Building final list result
		for (Object[] o : oneResutPerRoute) {
			// Getting route by name
			List<Routes> route =
em.createNamedQuery("Routes.findByName")
					.setParameter("name", o[0])
					.getResultList();
			// Getting start stop by id
			List<Stops> stop1 =
em.createNamedQuery("Stops.findByIdStop")
					.setParameter("idStop", o[1])
					.getResultList();
			// Getting end stop by id
			List<Stops> stop2 =
em.createNamedQuery("Stops.findByIdStop")
```

```
                        .setParameter("idStop", o[2])
                        .getResultList();
                RouteStops routeFinalitem = new RouteStops();
                routeFinalitem.setRoute(route.get(0));
                routeFinalitem.setStart_stop(stop1.get(0));
                routeFinalitem.setEnd_stop(stop2.get(0));
                routeFinal.add(routeFinalitem);
            }
            // Return result

            return routeFinal;
        } else {

            return null;
        }
    }
```

MANUAL DE USUARIO

Aplicación SITApp

Grupo de Investigación Aplicada en Sistemas

TABLA DE CONTENIDO

TABLA DE FIGURAS

MANUAL DE USUARIO

Título del software: SITApp

Este documento pretende guiar a los usuarios finales en el uso e interacción de la aplicación móvil denominada "SITApp: una aplicación inteligente para dispositivos móviles del sistema estratégico de transporte público del municipio de pasto". Esta app tiene como objetivo asociar sitios de interés de la ciudad de Pasto a una determinada ruta del Sistema Estratégico de Transporte Público del municipio de Pasto (SETP).

1. INTERFAZ PRINCIPAL

En la figura 1, se puede observar la interfaz principal del aplicativo móvil. Esta consta de varios elementos. Un menú desplegable (número 1), una barra de búsqueda (número 2) y una sección de mensajes (número 3). El sistema solicita activar el GPS y dar permisos de acceso a la ubicación del equipo puesto que la función principal del app hace uso del GPS para ubicar una ruta desde la ubicación del usuario hasta el sitio digitado sobre la barra de búsqueda.

El app hace uso de sitios de interés de la ciudad; un sitio de interés puede ser clínicas, hospitales, hoteles, centros comerciales, museos, bancos, parques, etc. Cabe destacar que se encuentran almacenados los sitios más relevantes de la ciudad.

Consideraciones, el sistema hace uso de paraderos autorizados por el SET P del municipio de Pasto, así que el app muestra el camino que el usuario debe hacer para llegar al paradero de inicio y además muestra el camino que el usuario debe recorrer para llegar a su destino desde el paradero de fin de viaje. En ocasiones la ruta seleccionada pasa justo por el sitio destino pero el sistema sugiere abandonar el bus antes o después del sitio destino esto es debido a que en el sitio destino no hay un paradero autorizado.

Figura 1. Interfaz principal SITApp

Fuente: Esta investigación, aplicación en ejecución.

A continuación se describen cada uno de los elementos de la interfaz principal.

2. SECCIÓN MENSAJES

Esta parte de la interfaz es utilizada para brindar al usuario final una pequeña ayuda en cada una de las funciones que tiene el aplicativo. Cada vez que el usuario selecciona una de las opciones disponibles en el menú desplegable la sección mensajes se actualiza y

muestra al usuario una ayuda referente a la opción seleccionada. Además esta sección es utilizada para mostrar información acerca de la ruta seleccionada para realizar el viaje en bus, ver figura 2.

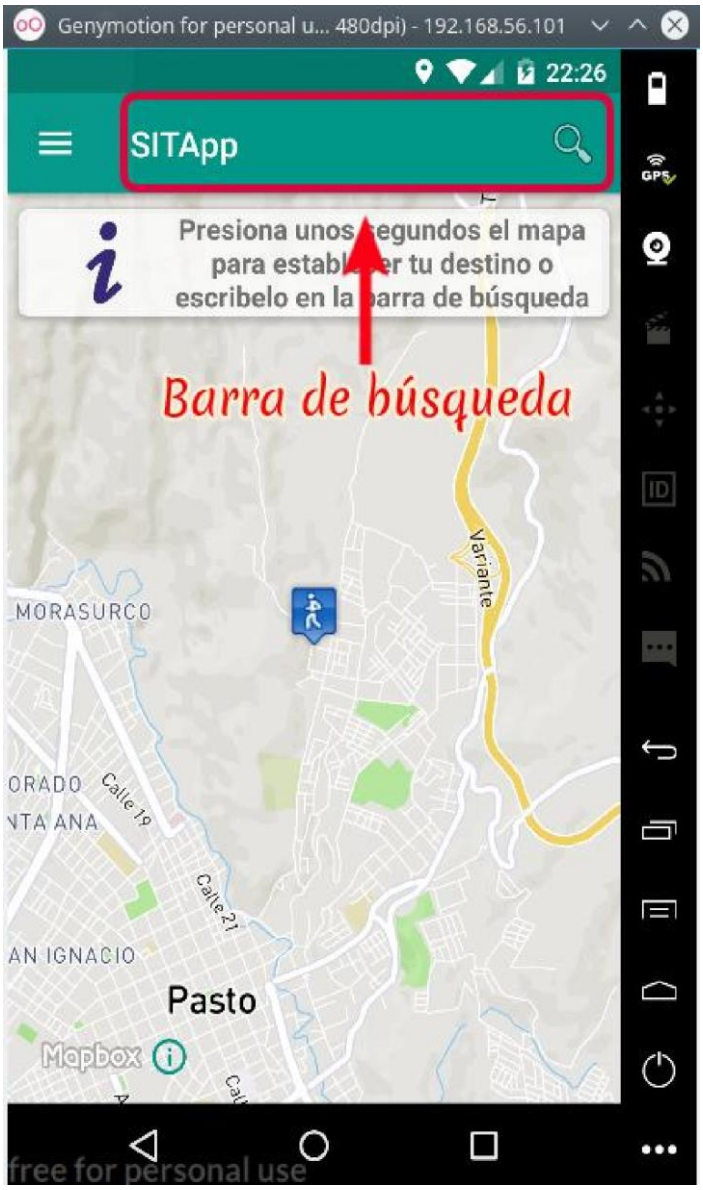
Figura 2. Sección de mensajes del aplicativo móvil

Fuente: Esta investigación, aplicación en ejecución.

3. BARRA DE BÚSQUEDA

Este elemento es utilizado para escribir el sitio de interés de la ciudad de Pasto. Esta sección hace que el aplicativo sea inteligente puesto que la barra de búsqueda, ver figura 3, hace uso de semántica para realizar las búsquedas en la base de datos, es decir, aun cuando existan errores de ortografía, el sistema intentará dar respuesta a las diferentes consultas realizadas por el usuario.

Figura 3. Barra de búsqueda del aplicativo SITApp

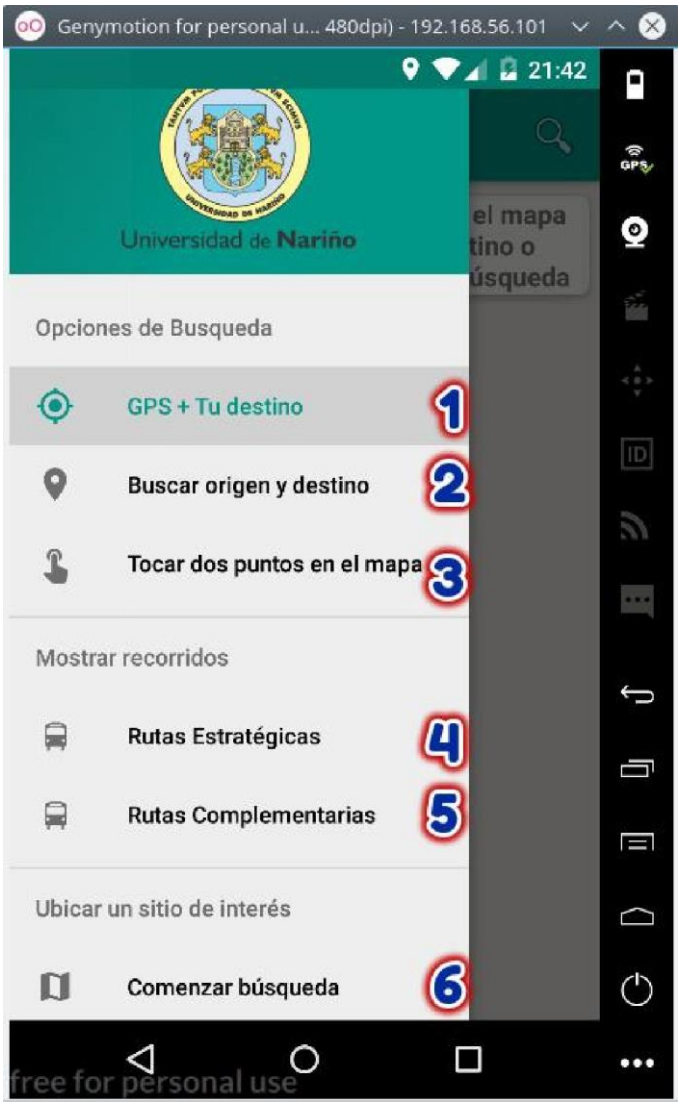


Fuente: Esta investigación, aplicación en ejecución.

4. MENÚ DESPLEGABLE

Este elemento despliega una sección desde la cual el usuario puede hacer uso de las funciones con las cuales cuenta SITApp, ver figura 4.

Figura 4. Menú desplegable SITApp



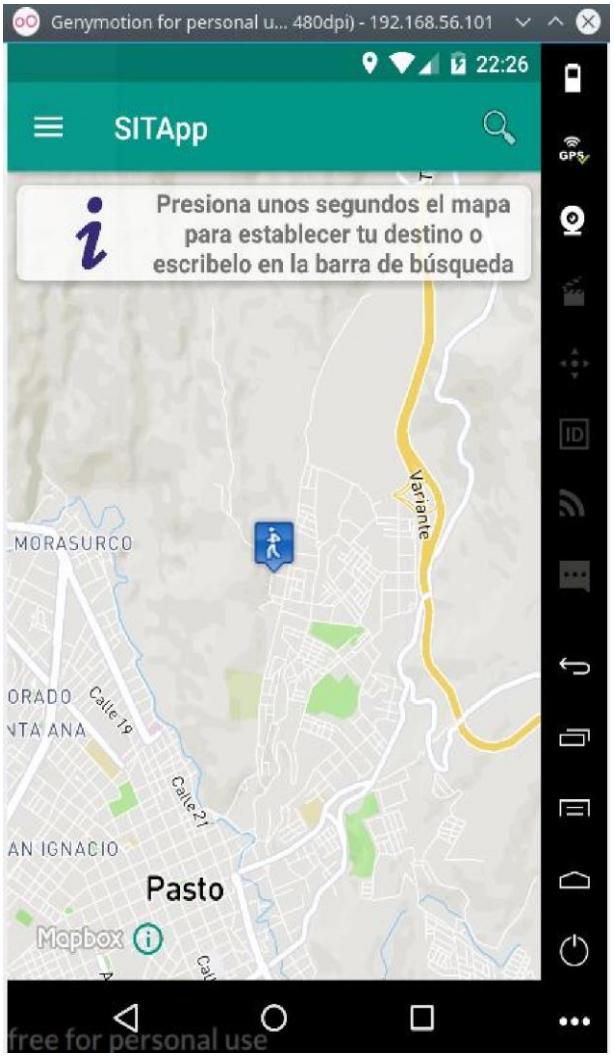
Fuente: Esta investigación, aplicación en ejecución.

A continuación se describen cada una de las funcionalidades que tiene SITApp.

4.1. GPS + Tu destino

Función principal del app y además es la función que está activa por defecto, es decir, cada vez que se abre el app, la función GPS + TU DESTINO está activa. Esta función hace uso del GPS del dispositivo desde el cual se está haciendo uso del aplicativo. El app utiliza el GPS y toma como punto de partida la ubicación del teléfono, el usuario debe escribir, en la barra de búsqueda, el sitio al cual quiere llegar o puede tocar el mapa unos segundos y el sistema realiza la búsqueda de rutas que pasen cerca del sitio donde se encuentra el usuario y que además pasen cerca del sitio destino, digitado o pulsado por el usuario, ver figura 5.

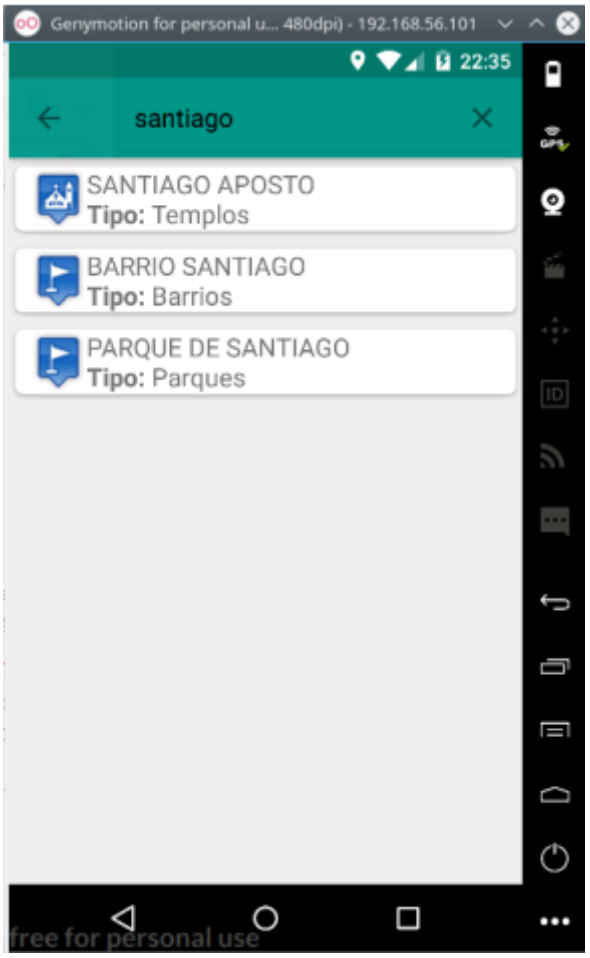
Figura 5. Uso de opción GPS + Tu destino.



Fuente: Esta investigación, aplicación en ejecución.

En la figura anterior, el icono que se puede observar en el mapa, hace referencia a la ubicación actual del usuario.

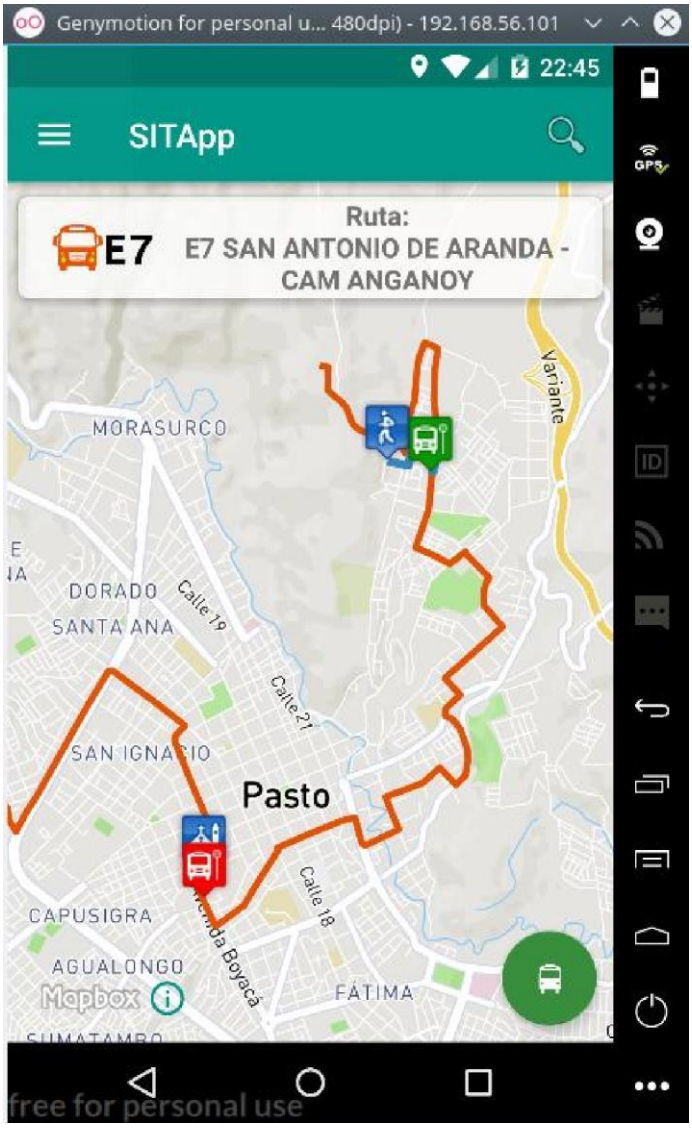
Figura 6. Uso barra de búsqueda para localizar destino.



Fuente: Esta investigación, aplicación en ejecución.

En la figura anterior, al hacer uso de la barra de búsqueda, el usuario debe seleccionar una de las opciones presentadas por el app y esta selección será tomada como destino y se procederá a realizar el cruce de rutas.

Figura 7. Resultado de búsqueda.



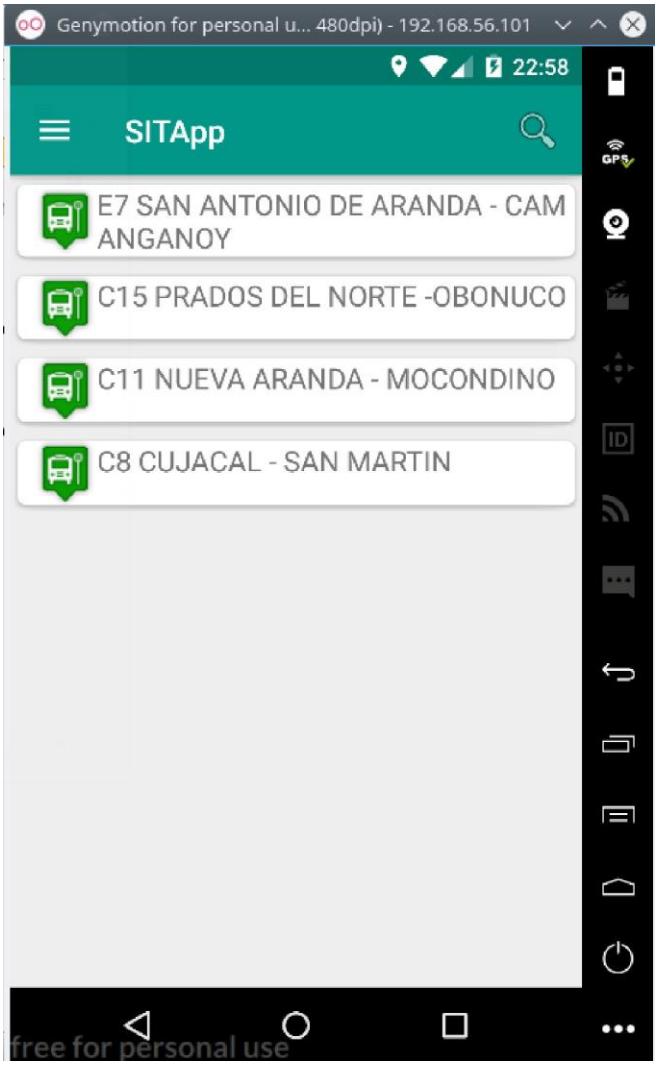
Fuente: Esta investigación, aplicación en ejecución.

Consideraciones para la figura anterior.

- El aplicativo, en la sección mensajes, muestra información de la ruta seleccionada para realizar el viaje para este ejemplo la ruta seleccionada es la E7.

- Los iconos se pintan de acuerdo al tipo de sitio, es decir, si es una iglesia entonces el icono corresponderá con una iglesia o si es un parque el icono se pintara con una imagen que haga alusión a este tipo de sitio.
- La línea de color naranja representa el recorrido que realiza el bus. • El icono de color verde hace referencia al paradero al cual se debe dirigir el usuario para tomar el bus.
- El icono de color rojo hace referencia al paradero en el cual el usuario debe bajarse.
- Las líneas de color azul, dibujan el camino que el usuario debe seguir para llegar a su sitio destino, en caso de que el bus no pase justo sobre el sitio destino.
- En caso de existir más de una ruta que pase cerca a la ubicación del usuario y que además pase cerca al sitio destino, entonces, el app muestra un botón de color verde, el cual está ubicado en la parte inferior derecha, al presionar este botón se despliega una lista de rutas que el usuario puede seleccionar para observar cual es el recorrido seguido y así tomar la ruta que él considere mejor, ver figura 8.

Figura 8. Lista de posibles rutas para realizar el viaje en bus.

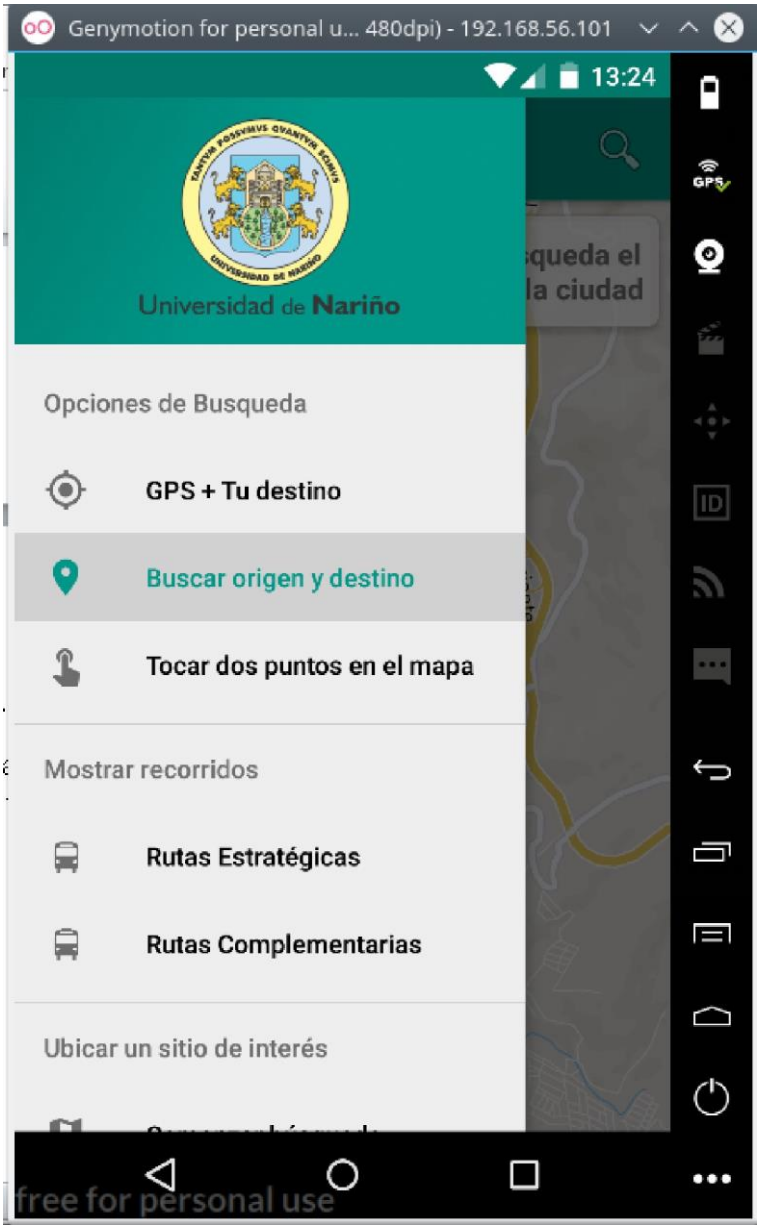


Fuente: Esta investigación, aplicación en ejecución.

4.2. Buscar origen y destino

Esta función permite que el usuario, haciendo uso de la barra de búsqueda, digite el sitio de origen y el punto destino, ver figura 9.

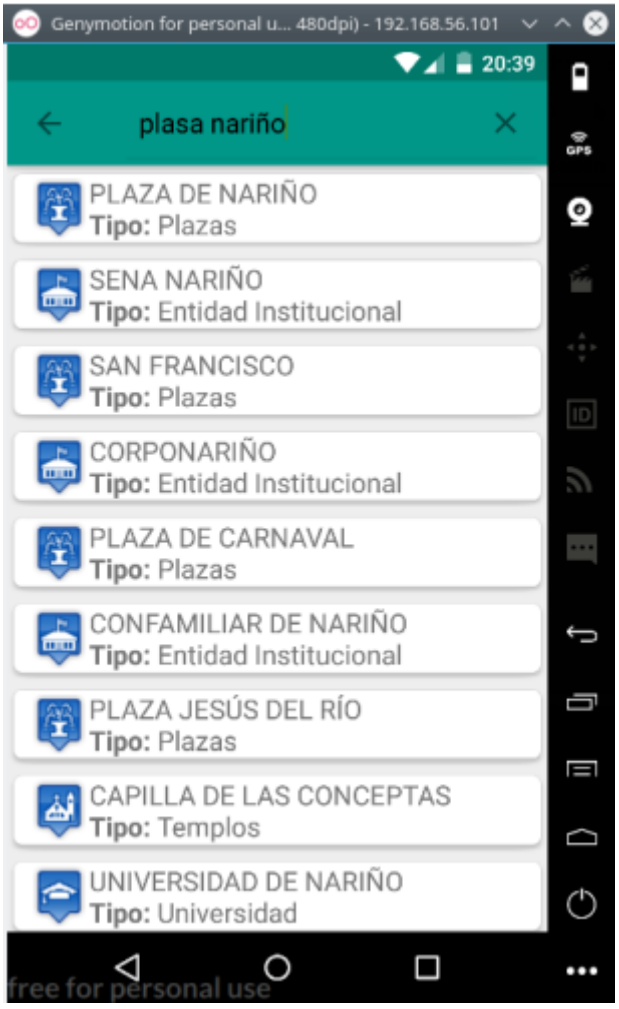
Figura 9. Función buscar origen y destino



Fuente: Esta investigación, aplicación en ejecución.

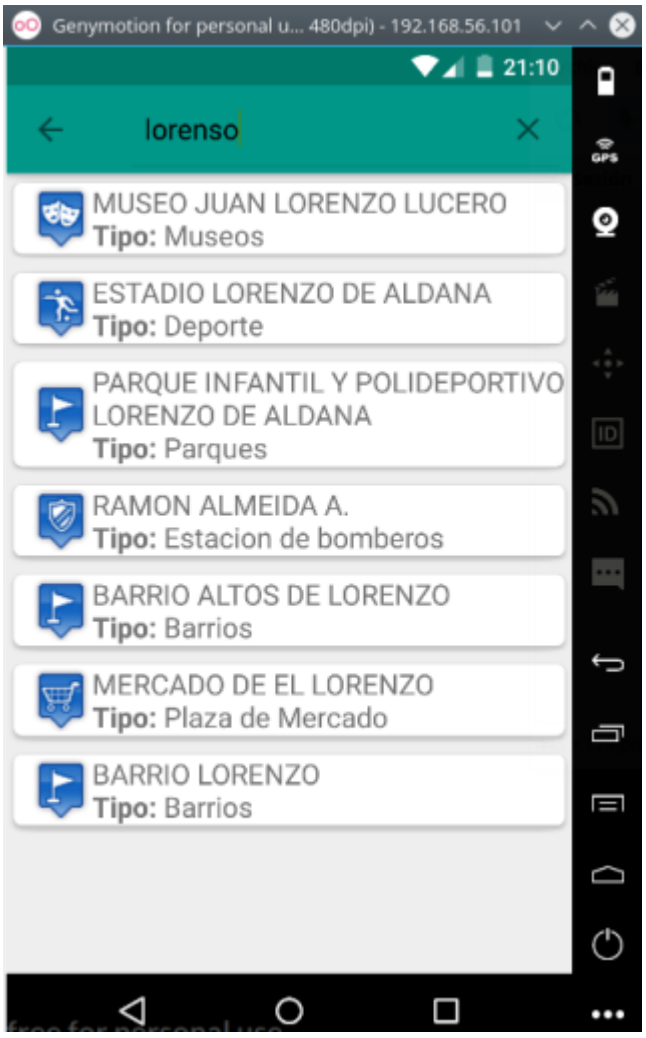
En este caso el usuario debe digitar los dos sitios, origen y destino, haciendo uso de la barra de búsqueda.

Figura 10. Buscar origen de viaje.



Fuente: Esta investigación, aplicación en ejecución.

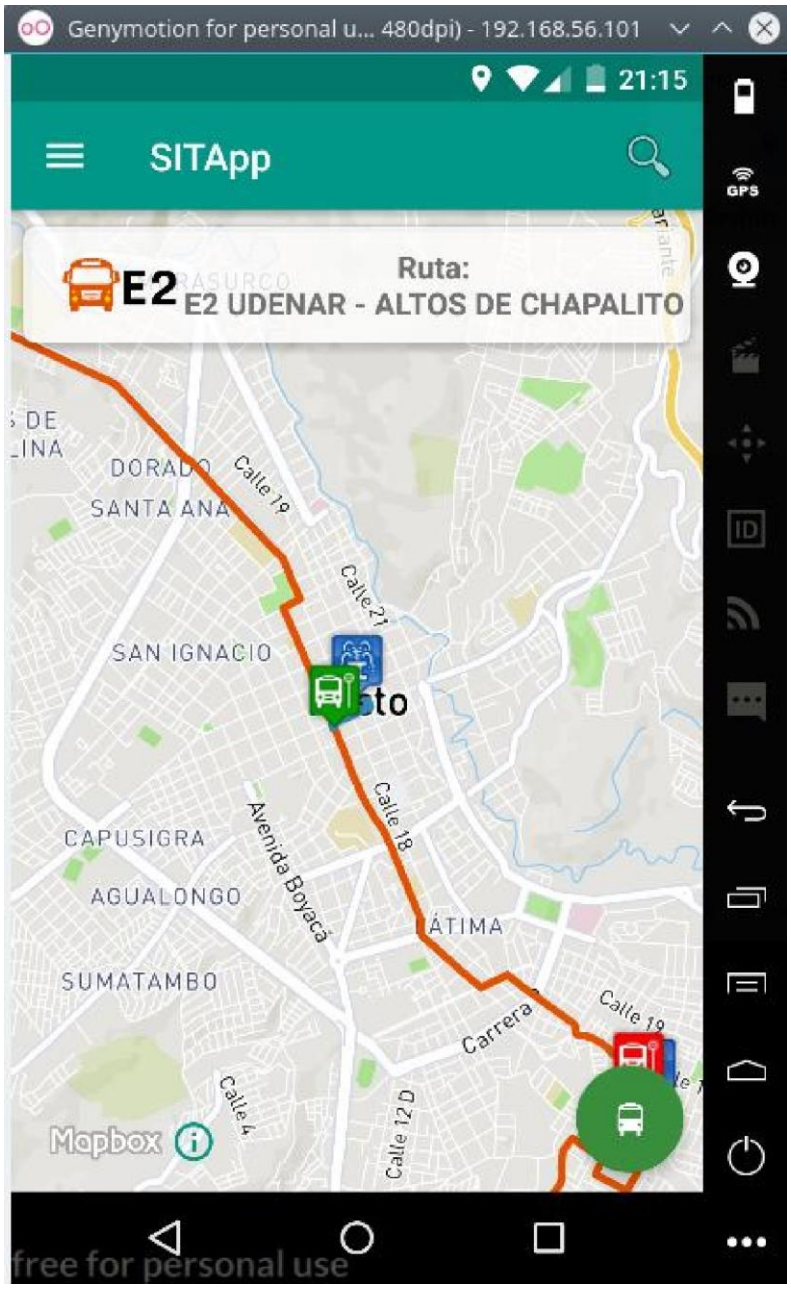
Figura 11. Buscar destino de viaje



Fuente: Esta investigación, aplicación en ejecución.

Como se puede observar en las dos figuras anteriores, a pesar de existir errores de ortografía el sistema brinda una seria de posibles respuestas.

Figura 12. Resultado opción Buscar origen y destino



Fuente: Esta investigación, aplicación en ejecución.

El ejemplo anterior es un viaje desde Plaza de Nariño hasta el barrio Lorenzo. Al igual que en la función GPS +TU DESTINO, en el caso de existir más de una ruta que cumpla con las condiciones de viaje el sistema muestra un botón de color verde el cual permite conocer más de una opción al momento de viajar en bus.

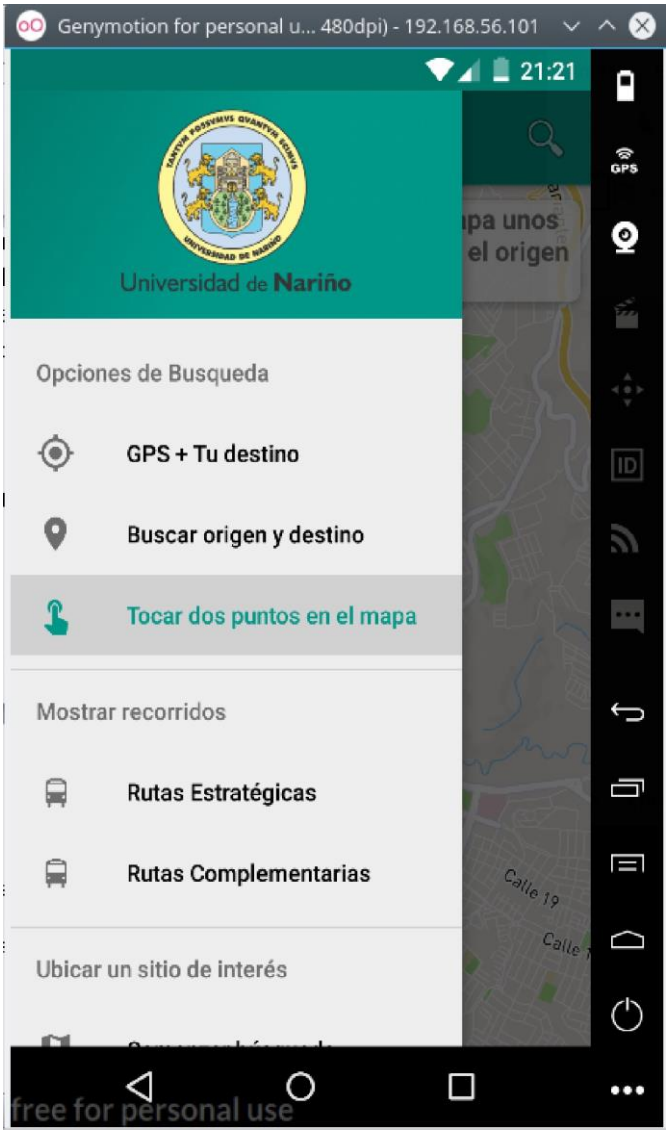
La sección mensajes ha sido actualizada y ahora muestra información de la ruta seleccionada para realizar el viaje en bus.

4.3. Tocar dos puntos en el mapa

Esta función permite que el usuario pueda hacer clic sobre el mapa y el punto que el usuario seleccione será tomado como parámetro en la búsqueda de rutas del SET P del municipio de Pasto.

En este caso el usuario debe presionar unos segundos el mapa y al momento de que se dibuje el marcador hacer clic sobre la ventana emergente, ver figura 13.

Figura 13. Función Tocar dos puntos en el mapa.



Fuente: Esta investigación, aplicación en ejecución.

Figura 14. Tocar mapa para seleccionar origen de viaje.

Fuente: Esta investigación, aplicación en ejecución.

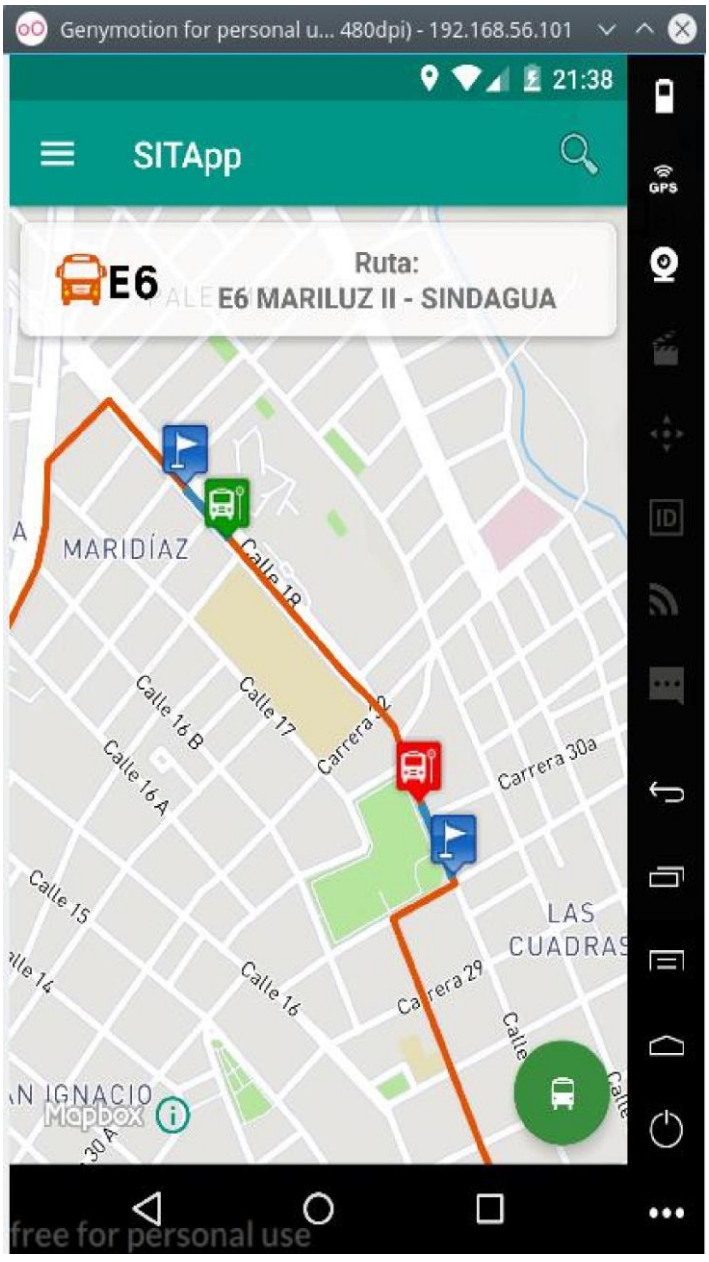
En la figura anterior, se puede apreciar que el sistema muestra un mensaje indicando al usuario que debe hacer, además se muestra un icono de color azul, el cual tiene una ventana sobre él, indicando que se debe presionar este elemento para establecer el origen del viaje.

Figura 15. Tocar puntos para establecer destino de viaje.

Fuente: Esta investigación, aplicación en ejecución.

En la figura anterior se puede ver que la sección mensajes fue actualizada para indicar al usuario que debe ubicar su destino, además se dejó ubicado el origen del viaje para que el usuario pueda observar cual su origen y cual será su destino. Al igual que en el paso anterior, el usuario debe presionar el elemento ubicado sobre el marcador para establecer el destino de viaje.

Figura 16. Resultado Tocar dos puntos en el mapa



Fuente: Esta investigación, aplicación en ejecución.

Al igual que en las dos funciones anteriores el sistema muestra paradero de inicio de viaje y paradero de fin de viaje además del botón que permite conocer más opciones de rutas que cumplan con las condiciones de viaje.

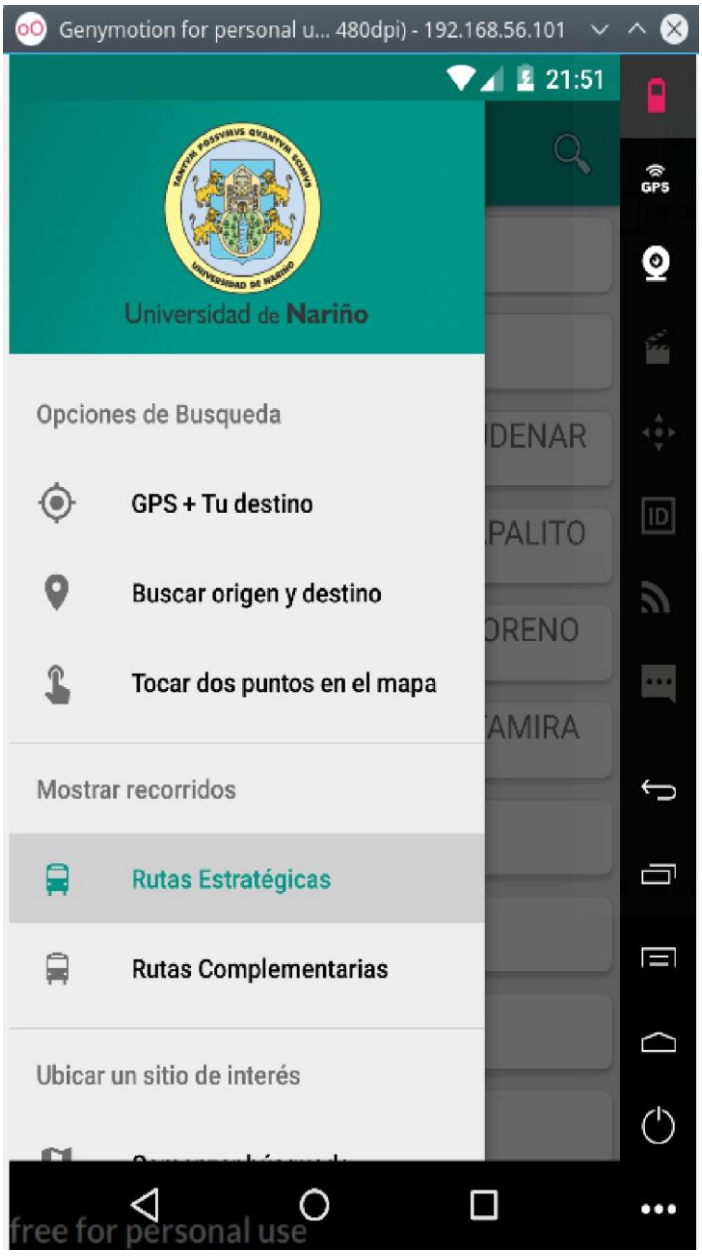
El espacio para mensajes ha sido actualizado y ahora muestra información de la ruta seleccionada para realizar el viaje en bus.

4.4. Mostrar recorridos de rutas.

Esta opción permite que el usuario pueda ver el recorrido de las rutas y las diferentes paradas que hace el bus.

En esta parte el usuario únicamente debe seleccionar la ruta, de la cual desea ver el recorrido, y el sistema mostrará sobre el mapa el recorrido y las paradas que hace la ruta seleccionada, ver figura 17.

Figura 17. Opción Recorrido de rutas.



Fuente: Esta investigación, aplicación en ejecución.

Figura 18. Ruta C9, recorrido y paradas.

Fuente: Esta investigación, aplicación en ejecución.

Consideraciones.

- La sección mensajes muestra detalles de la ruta seleccionada.
- Sobre el mapa se pinta recorrido de ruta, color naranja, paradas, color azul.

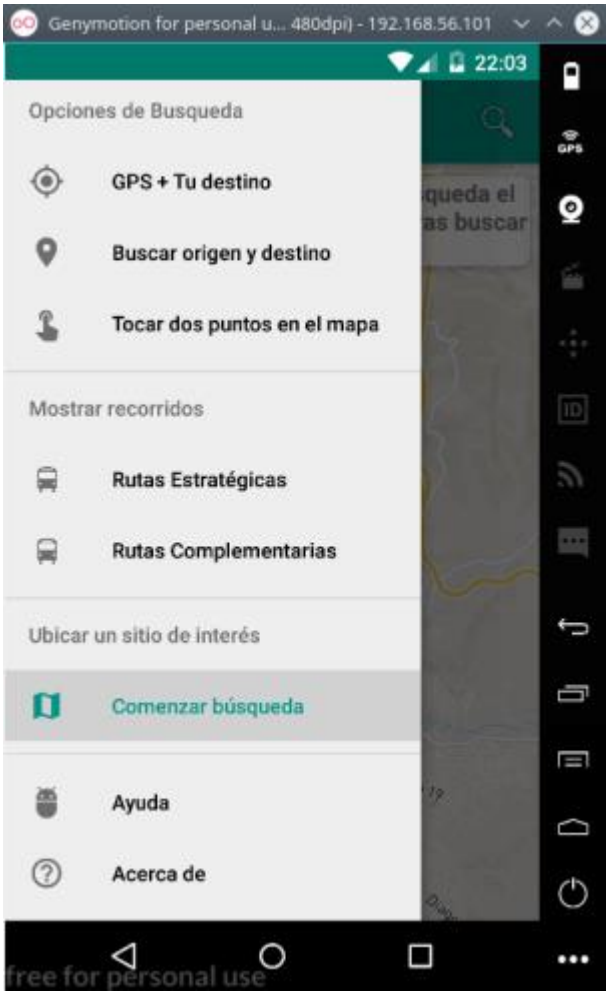
- Iconos de color verde y rojo indican, respectivamente, inicio y fin de recorrido.
- El usuario puede hacer clic sobre una de las paradas para conocer algunos detalles de la parada seleccionada.

4.5. Ubicar sitio de interés

Esta opción permite encontrar sitios de interés de la ciudad de Pasto.

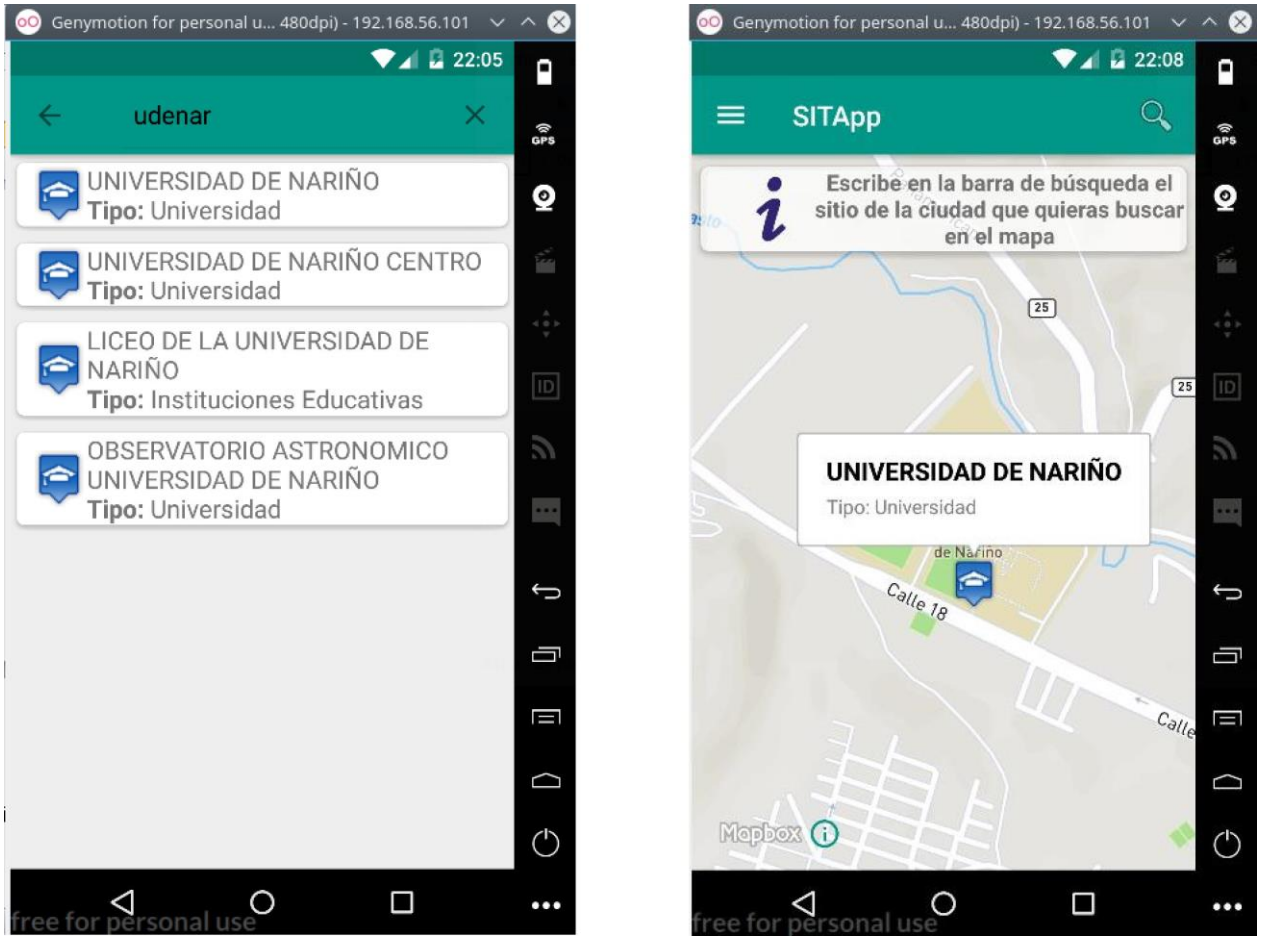
Al igual que en la función ORIGEN Y DESTINO, el usuario debe hacer uso de la barra de búsqueda para encontrar el sitio deseado, ver figura 19.

Figura 19. Función Ubicar sitio de interés



Fuente: Esta investigación, aplicación en ejecución.

Figura 20. Ubicar sitio de interés de la ciudad de Pasto.



Fuente: Esta investigación, aplicación en ejecución.

MANUAL DE PROGRAMADOR

Módulo Administrativo

Grupo de Investigación Aplicada en Sistemas

TABLA DE CONTENIDO

TABLA DE FIGURAS

MANUAL DE PROGRAMADOR MÓDULO ADMINISTRATIVO SITApp

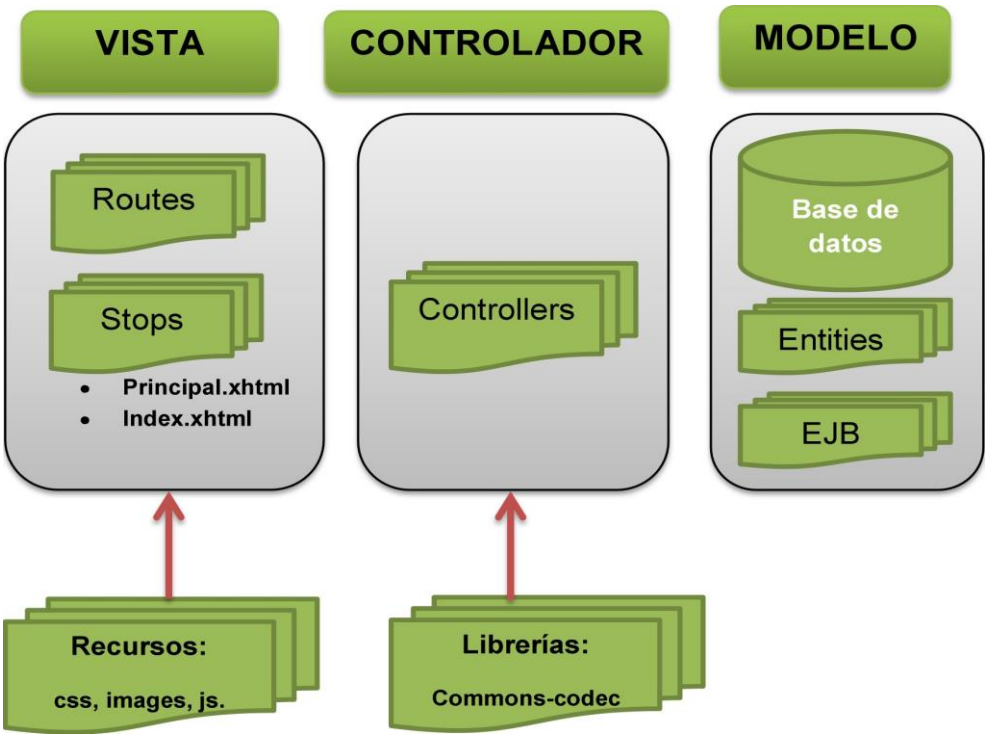
El presente documento es una guía para los programadores que hacen uso del módulo administrativo de la aplicación móvil denominada SITApp, es un manual práctico para la compresión y descripción de las principales funcionalidades que tiene este módulo.

La construcción del módulo administrativo fue realizada en su totalidad bajo el sistema operativo Linux Mint 18.1 KDE Edition. El lenguaje de programación utilizado fue Java, en específico se hizo uso del framework Java Server faces (jsf), primefaces, javascript, css, omnifaces, commons-codec y mapbox para el manejo de mapas, el entorno de desarrollo empleado fue Netbeans versión 8.1. El sistema está montado sobre el servidor de aplicaciones Glassfish versión 4.1.

1. ARQUITECTURA

Este módulo está desarrollado bajo el modelo MVC (Modelo Vista Controlador). En la Figura 1, se puede observar la arquitectura.

Figura 1. Arquitectura módulo administrativo SITApp



Fuente: Esta investigación.

A continuación se describe cada uno de los elementos de la arquitectura del módulo administrativo de SITApp.

- Vista.

Permite la interacción del usuario con el usuario final, en donde se presenta la vista con el menú de opciones para las distintas opciones disponibles, los datos ingresados por el usuario son procesados por el paquete controlador y los resultados se muestran al usuario haciendo uso de etiquetas de primefaces y un mapa de mapbox. Este paquete esta compuesto por páginas con extensión xhtml de primefaces y hace uso de algunos recursos como imágenes, archivos css, javascript, además este paquete hace uso de una librería llamada omnifaces la cual permite la transformación de datos primitivos a objetos java.

- Controlador.

Contiene la lógica de negocios de la aplicación. Se encarga de validar los datos ingresados por el usuario y procesarlos para dar respuesta a las diferentes peticiones. Permite la conexión con la base de datos y hace uso de métodos especializados para realizar operaciones CRUD (Create, Read, Update and Delete) en la base de datos. Por cada archivo con extensión xhtml existe una clase Java que gestiona las peticiones realizadas por esa vista. Además este paquete hace uso de una librería llamada commons-codec, la cual es utilizada para encriptar la contraseña del usuario.

- Modelo.

Esta compuesto por dos paquetes.

- Entities: Contiene el modelo de la base de datos representado en clases escritas en lenguaje de programación Java, las cuales emulan cada una de las tablas de la base de datos. El mapeo de las tablas, con todos sus atributos y restricciones, lo realiza el framework JPA (Java Persistence Api).
- EJB: Contiene un conjunto de clases que se encargan de realizar el acceso a los registros contenidos en la base de datos, al momento de intentar realizar cualquier operación sobre la base de datos, insertar, leer, actualizar o eliminar registros, se debe hacer uso de este paquete.

2. FUNCIONALIDADES DEL MÓDULO ADMINISTRATIVO.

A continuación se describen las funcionalidades del módulo administrativo del aplicativo móvil denominado SITApp.

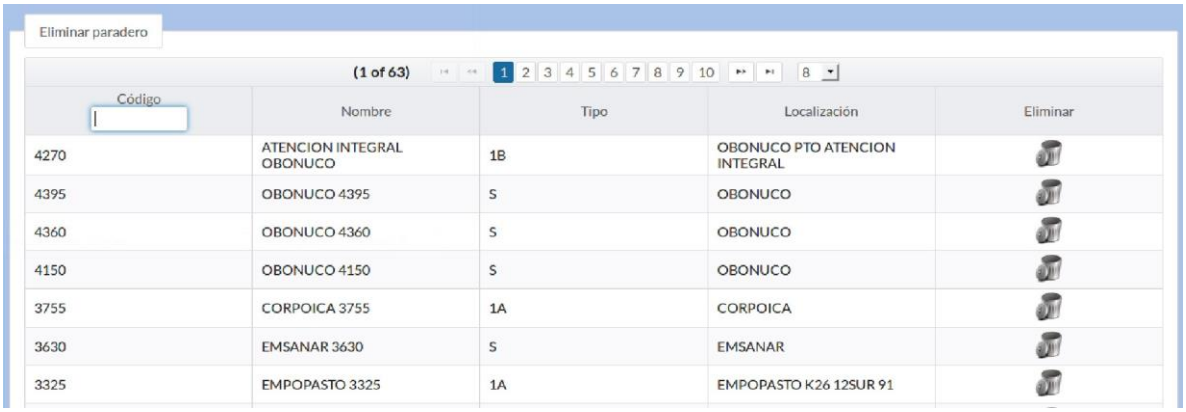
Cabe destacar que por cada archivo con extenxión .xhtml existe una clase java y un archivo JavaScript, esto con el fin de dar solución a las peticiones realizadas por el cliente hacia el servidor y además lograr mostrar el resultado sobre el navegador.

2.1. Eliminar paradero

En esta función el sistema muestra un listado de todos los paraderos, que tiene el Sistema Estratégico de Transporte Público del municipio de Pasto (SETP), el usuario debe seleccionar el paradero que desea eliminar. Cuando el usuario selecciona un elemento para ser eliminado se despliega un dialogo con información del paradero seleccionado y dos botones, Eliminar o Cancelar.

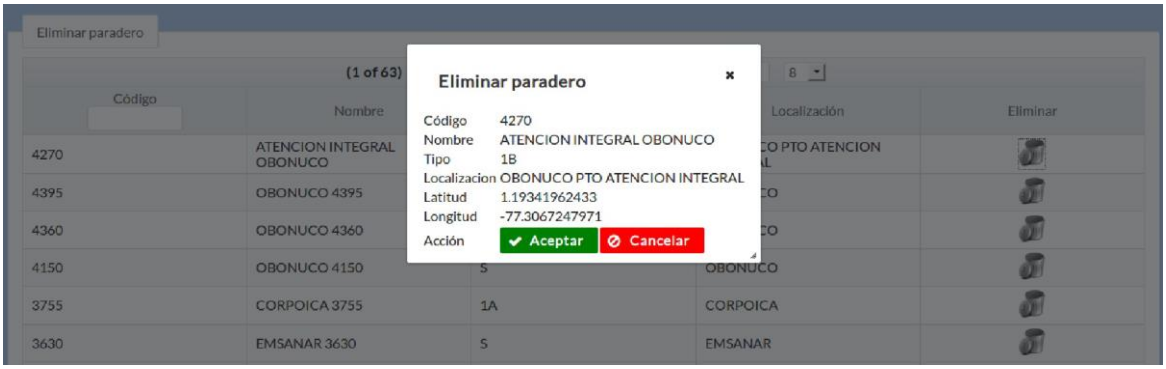
Los elementos incorporados en la vista son un dataTable, un dialog y tres commandButton, todos ellos elementos de primefaces.

Figura 2. Eliminar paradero del SETP del municipio de Pasto.



Fuente: Esta investigación.

Figura 3. Confirmación de eliminación de paradero.



Fuente: Esta investigación.

La selección pasa al controlador ListStopsController.java, esta clase se encarga de gestionar la eliminación de paraderos del SETP. Al momento que el usuario presiona el botón Aceptar, se ejecuta el método deleteStop(), el cual elimina el paradero y todas las relaciones que este tenga con alguna ruta del SET P. En caso de presionar el botón Cancelar, el dialogo se cierra. A continuación, se presenta un fragmento del código que realiza la tarea de eliminar un paradero de la base de datos.

```
try {
    lstStopsRoutes = EJBStopsRoutes.findRouteByIdStop(stop.getIdStop());
    if (lstStopsRoutes != null) {
       for (StopsRoutes sr : lstStopsRoutes) {
          EJBStopsRoutes.updateSequencesWhenDisassociate(
                sr.getStopsRoutesPK().getRouteName(),
                sr.getStopsRoutesPK().getSecuence());
       }
    }
    /**
     * This instruction Delete stop of the data base.
     */
    EJBStop.remove(stop);
    this.setStops(EJBStop.findAll());
    messageController.infoMessage("Paradero Eliminado");

 } catch (Exception e) {

messageController.errorMessage("Paradero NO Eliminado " + e.getMessage());
 }
```

Una vez eliminado el paradero se actualiza el array de paraderos y el dataTable que contiene esta información. Al finalizar todo el proceso de eliminación del paradero se muestra el mensaje respectivo al usuario final.

2.2. Crear paraderos

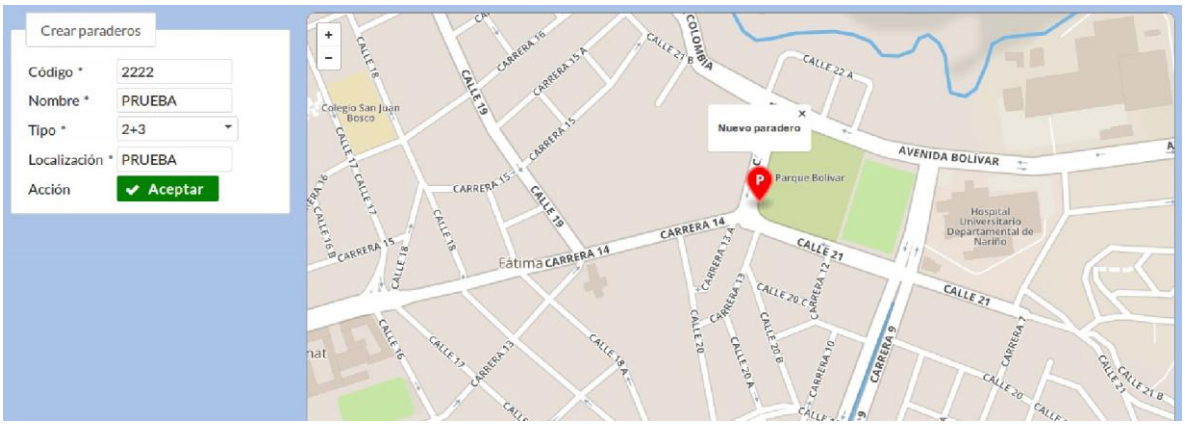
Esta función permite crear nuevos paraderos en el SET P del municipio de Pasto. El usuario debe ingresar la información solicitada, todos los datos solicitados son obligatorios, además el usuario debe hacer uso del mapa para ubicar el paradero, en la figura 3 se puede observar

el formulario para la creación de paraderos. Para pintar el paradero sobre el mapa, el usuario debe hacer doble clic sobre este elemento.

Los campos solicitados surgieron de la información entregada por AVANTE.

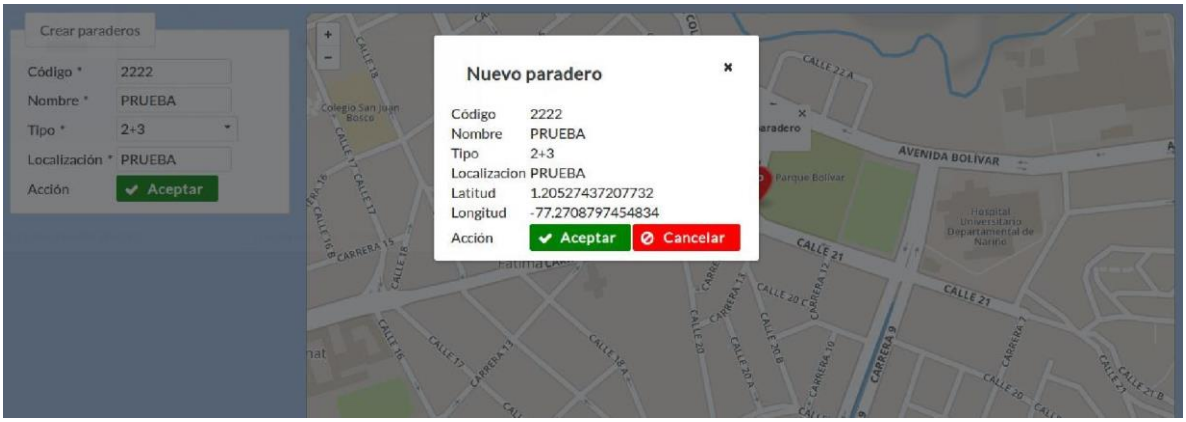
Los elementos presentes en el formulario para creación de paraderos son fielset, un panelGrid, cinco outputLabel, tres inputText, un selectOneMenu, un dialog, tres botones y un mapa, todos estos elementos son etiquetas de primefaces a excepción del mapa, este elemento es proporcionado por MapBox.

Figura 4. Formulario para la creación de paraderos en el SETP.



Fuente: Esta investigación.

Figura 5. Dialogo con información del nuevo paradero.



Fuente: Esta investigación.

Los datos ingresados por el usuario son enviados al controlador CreateStopsControIIer.java, esta clase se encarga de crear y almacenar los nuevos paraderos en la base de datos. En el momento en que el usuario presiona el botón Aceptar, el cual está dentro del dialogo de confirmación, se ejecuta el método createStops(). A continuación se muestra un fragmento de código, el cual crea y almacena los nuevos paraderos.

```
try {
    stop.setName(stop.getName().toUpperCase());
    stop.setTypeStop(stop.getTypeStop().toUpperCase());
    stop.setLocalization(stop.getLocalization().toUpperCase());
    EJBStop.create(stop);
    this.listStopsController.setStops(EJBStop.findAll());

    messageController.infoMessage("Paradero creado");

} catch (Exception e) {
    messageController.errorMessage("Paradero NO creado");
}
```

Para pintar el paradero sobre el mapa se hace uso de JavaScript. A continuación se muestra la sentencia encargada de dibujar el paradero sobre el mapa.

```
markerStop = L.marker(L.latLng(latitude, longitude), {
    title: 'Nuevo Paradero',
    draggable: true,
    icon: L.mapbox.marker.icon({
        'marker-color': '#FF0000',
        'marker-symbol': 'parking',
        'marker-size': 'large'
    })
})
        .bindPopup("<strong>Nuevo paradero</strong>")
        .addTo(map);
```

Después de crear y almacenar el nuevo paradero se procede a actualizar el array de paraderos y el dataTable que contiene esta información.

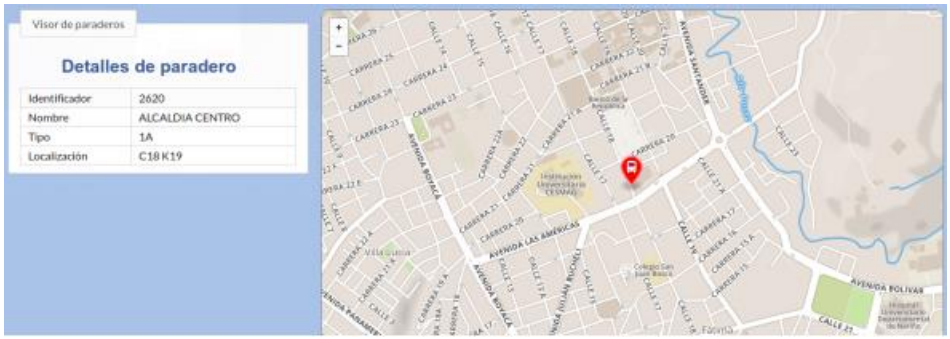
2.3. Ver y editar

Esta función permite visualizar o editar un paradero del SET P del municipio de Pasto. El sistema despliega un listado de todos los paraderos del SETP y además en la parte derecha de cada registro se muestran dos botones con las opciones Ver y Editar.

Para esta vista se hizo uso de un fielSet, un dataTable y dos commandButtons, estos elementos son de primefaces.

- VER: Permite visualizar la información de un paradero seleccionado y además se muestra el paradero sobre un mapa, ver figura 6.

Figura 6. Visualización de paradero 2620, ALCALDIA CENTRO.



Fuente: Esta investigación.

Los elementos presentes en la vista de paraderos son un fielset, un hl , ocho outputLabel estos son elementos de primefaces a excepción del hl, el cual es un elemento de HTML, además se tiene un mapa el cual es proporcionado por MapBox. El paradero seleccionado pasa al controlador ViewStopControIler.java. Esta clase se encarga de enviar los datos del elemento hacia la vista de paraderos. En cuanto se presiona el botón ver, en el listado de paraderos, se re-direcciona a la vista de paraderos, la cual muestra toda la información del paradero además de su ubicación en el mapa. A continuación se presenta un fragmento de código para mostrar el paradero.

```
var marker = L.marker(L.latLng(latitud, longitud),
{
   icon: L.mapbox.marker.icon({
      'marker-color': '#FF0000',
      'marker-symbol': 'bus',
      'marker-size': 'large'
   })
})
.bindPopup(nombre).addTo(map);
```

- EDITAR: Permite actualizar la información de un paradero. Al igual que en la función VER, se muestra la información del paradero pero con la posibilidad de que esta sea

editada. En la vista se puede observar los detalles del paradero así como también el lugar donde está ubicado en el mapa, ver figura 7.

Figura 7. Edición de paradero 3755, CORPOICA.

Fuente: Esta investigación.

En esta vista se hizo uso de un fieldSet, 5 outputLabel, 5 inputText y un botón, estos son elementos de primefaces. Además se hizo uso de un hl ,elemento de HTML, y un mapa proporcionado por MapBox.

Cuando el usuario presiona el botón Aceptar, la información del paradero es enviada al controlador EditStopControIIer.java, y se ejecuta el método editStop(). A continuación se muestra un fragmento de código para la edición del paradero.

```
        try {
            stop.setName(stop.getName().toUpperCase());
            stop.setTypeStop(stop.getTypeStop().toUpperCase());
            stop.setLocalization(stop.getLocalization().toUpperCase());
            EJBStop.edit(stop);
            this.listStopsController.setStops(EJBStop.findAll());

            messageController.infoMessage("Paradero actualizado");

        } catch (Exception e) {
            messageController.errorMessage("Paradero NO actualizado");
         }
        }
```

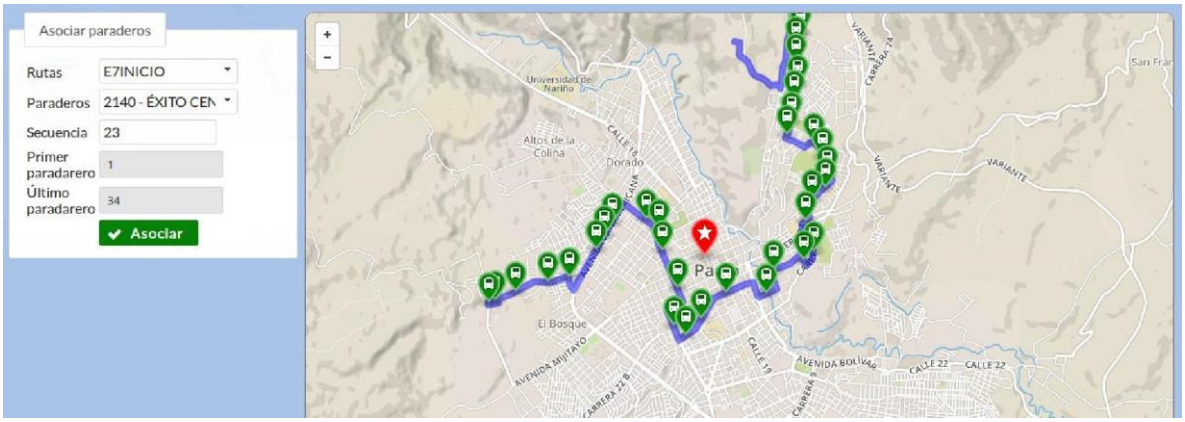
Además el paradero se muestra sobre un mapa, así que se hizo uso de JavaScript para realizar esta acción. A continuación se muestra un fragmento de código encargado de ubicar el paradero sobre el mapa. Para editar la ubicación del paradero solo es necesario mover el marcador hacia la posición deseada.

```
    var marker = L.marker(L.latLng(latitud, longitud),
        {
            draggable: true,
            icon: L.mapbox.marker.icon({
                'marker-color': '#FF0000',
                'marker-symbol': 'parking',
                'marker-size': 'large'
            })
        })
        .addTo(map);
    /**
     * when marker is draged its new coordinates are assign at the form's
components
     */
    marker.on('dragend', function () {
        latitud = marker.getLatLng().lat;
        longitud = marker.getLatLng().lng;
        document.getElementById("frmEditS:latitude").value = latitud;
        document.getElementById("frmEditS:longitude").value = longitud;
    });
```

2.4. Asociar paraderos

Esta función permite asociar paraderos a una ruta del SET P del municipio de Pasto, ver figura 8.

Figura 8. Asociar paraderos del SETP.



Fuente: Esta investigación.

Esta vista contiene un fieldSet, 8 outputLabel, dos selectOneMenu, tres inputText, dos de ellos de solo lectura, estos elementos son de primefaces y un mapa proporcionado por MapBox.

El usuario debe seleccionar la ruta y el paradero que desea asociar. El resultado de la sección se puede ver en un mapa donde el nuevo paradero se pinta de color rojo y los paraderos que ya forman parte de la ruta se pintan de color verde, además, se traza el recorrido del bus con una línea de color azul.

Los elementos seleccionados así como la secuencia son enviados al controlador, AssociateStopsControlIer.java, el cual se encarga de asociar los paraderos a la ruta seleccionada. A continuación se muestra un fragmento de código que hace esta tarea.

```
try {
            EJBStopsRoutes.updateSequencesWhenAssociate(route.getName(),
sequence);
            EJBStopsRoutes.create(new StopsRoutes(route.getName(),
stop.getIdStop(), sequence));
            setFirstSecuence(EJBStopsRoutes.firstStop(route.getName()));
            setLastSecuence(EJBStopsRoutes.lastStop(route.getName()));
            setStops(EJBStop.findNews(route.getName()));

            messageController.infoMessage("PARADERO ASOCIADO");
```

Una validación que se realiza en esta parte corresponde a la distancia entre la geometría de la ruta y el paradero. La distancia máxima entre estos dos elementos no debe superar los 2 metros, con esto garantizamos que el paradero que se desea asociar a esa ruta si corresponde. A continuación se muestra la consulta sql para realizar esta tarea.

```
String consult = "SELECT st_distance(st_geomfromtext(?1), st_point(?2,
?3))*10000 "
            + "FROM routes "
            + "WHERE routes.name = ?4";
```

Esta consulta se puede realizar gracias al uso de Postgis. Los parámetros que se le pasa a la consulta corresponden a la geometría de la ruta seleccionada, las coordenadas, latitud y longitud, del paradero seleccionado. Esta consulta retorna la distancia entre la geometría y los dos puntos enviados como parámetro.

Al igual que en las funciones anteriores, en esta también se hizo uso de JavaScript para mostrar resultados sobre el navegador. A continuación se muestra un fragmento de código que permite visualizar los datos en el navegador.

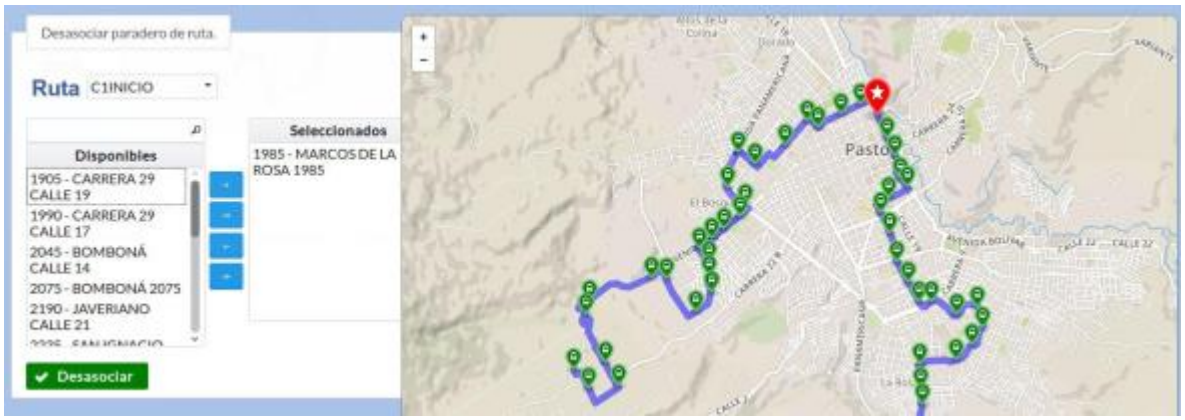
```
if (geometry !== "") {
    for (var i = 0; i < geometry.length - 1; i = i + 2) {
        var latlon = L.latLng(geometry[i], geometry[i + 1]);
        points.push(latlon);
    }
    polyline = L.polyline(points, {
        color: 'blue',
        opacity: 0.5,
        weight: 8
    }).addTo(map);
}
```

```
    counter = 0;
    /**
     * Drawing new stop
     */
    latlng = L.latLng(coordinates[0], coordinates[1]);
    marker = L.marker(latlng, {
        icon: L.mapbox.marker.icon({
            'marker-symbol': 'star',
            'marker-color': '#FF0000',
            'marker-size': 'large'
        })
    }).bindPopup("<strong>NUEVO PARADERO<br> secuencia: " + secuence +
"</strong>")
            .addTo(map);
    markers.push(marker);
    /**
     * Drawing stops associates to route
     */
    for (var i = 0; i < dataStops.length - 1; i = i + 3) {
        latlng = L.latLng(dataStops[i], dataStops[i + 1]);
        marker = L.marker(latlng, {
            icon: L.mapbox.marker.icon({
                'marker-symbol': 'bus',
                'marker-color': '#088A08'
            })
        }).bindPopup("<strong>" + dataStops[i + 2] + "<br>Secuencia: " +
lstSecuence[counter] + "</strong>")
                .addTo(map);
        markers.push(marker);
        counter = counter + 1;
    }
```

2.5. Desasociar paraderos

Esta función permite desasociar paraderos de una ruta del SET P. En esta opción el usuario debe seleccionar la ruta y el sistema muestra todos los paraderos asociados a esa ruta en específico. El usuario debe seleccionar el paradero a desasociar y presionar el botón Desasociar, ver figura 9.

Figura 9. Desasociar paraderos ruta Cl INICIO.



Fuente: Esta investigación.

Esta vista contiene elementos, de primefaces, tales como un fieldSet, un pickList y un commandButton, además se utilizó un hl, elemento de HTML, y un mapa de MapBox.

La ruta y los paraderos asociados se muestran sobre el mapa y cada vez que el usuario selecciona un paradero este es resaltado de color rojo y con un tamaño superior a los demás.

La información de los paraderos seleccionados es enviada al controlador, AssociateStopsControIler.java, que es la clase encargada de gestionar este tipo de peticiones. Dentro de esta clase se encuentra un método llamado dissasociateStops(), el cual da respuesta a la petición hecha por el usuario. A continuación se presenta un fragmento del código que realiza esta tarea.

```
        try {
            sequence =
Integer.parseInt(EJBStopsRoutes.findSecuence(route.getName(),
stop.getIdStop()));
```

```
                EJBStopsRoutes.remove(new StopsRoutes(route.getName(),
stop.getIdStop(), sequence));

EJBStopsRoutes.updateSequencesWhenDisassociate(route.getName(),
sequence);
                messageController.infoMessage("Paradero eliminado");
            } catch (Exception e) {
                messageController.errorMessage(e.toString());
            }
```

Para mostrar la ruta, los paraderos y además para resaltar el paradero seleccionado se hizo uso de JavaScript. A continuación se muestra un fragmento de código.

```
    /*
     * draw polyline
     */
    for (var i = 0; i < geometry.length - 1; i = i + 2) {
        var latlon = L.latLng(geometry[i], geometry[i + 1]);
        points.push(latlon);
    }
    polyline = L.polyline(points, {
        color: 'blue',
        opacity: 0.5,
        weight: 8
    }).addTo(map);
    /*
     * draw stops
     */

    if (state === false) {
        for (var i = 0; i < dataStops.length - 1; i = i + 4) {
```

```
        var latlng = L.latLng(dataStops[i], dataStops[i + 1]);
        marker = L.marker(latlng, {
           icon: L.mapbox.marker.icon({
              'marker-symbol': 'bus',
              'marker-color': '#088A08',
              'marker-size': 'small'
           })
        }).bindPopup("<strong>" + dataStops[i + 2] + " <br>  " + dataStops[i + 3] +
"</strong>")
              .addTo(map);
        markers.push(marker);
         }
```

2.6. Eliminar rutas

En esta opción el sistema muestra un listado de todas las rutas que tiene el SET P de la ciudad de Pasto. A la derecha de cada registro se muestra un botón, el cual permite eliminar una ruta del SET P, ver figura 10.

Figura 10. Eliminar ruta E2 INICIO del SETP.

Fuente: Esta investigación.

Cuando el usuario presiona el botón, con forma de bote de basura, se despliega un dialogo con información de la ruta seleccionada, ver figura 11.

Figura 11. Dialogo con información de ruta E2 INICIO.

Fuente: Esta investigación.

La información de la ruta seleccionada es enviada al controlador, ListRoutesController.java, clase encargada de gestionar la eliminación de rutas. Dentro de esta clase hay un método llamado deleteRoute() el cual elimina una ruta de la base de datos del SET P del municipio de Pasto. A continuación se presenta un fragmento del código encargado de eliminar una ruta.

```
try {
        EJBRoutes.remove(route);
        this.setRoutes(EJBRoutes.findAll());

        messageController.infoMessage("Ruta Eliminada");

    } catch (Exception e) {

        messageController.errorMessage("Ruta NO Eliminada");
    }
```

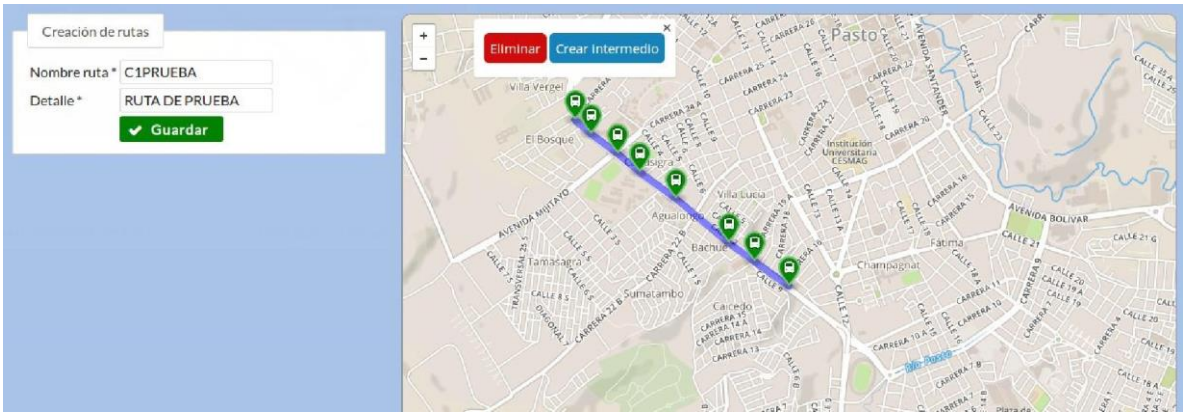
Después de eliminar la ruta se procede a actualizar el array y el dataTable que contiene la información de rutas del SETP.

2.7. Crear ruta

Esta función permite la creación de nuevas rutas en el SETP del municipio de Pasto. El usuario debe diligenciar los datos solicitados en el formulario para la creación de rutas, ver figura 12.

Este formulario fue creado utilizando dos outputLabel, dos inputText, un commandButton, etiquetas de primefaces, y un mapa de Mapbox.

Figura 12. Creación de rutas en el SETP.



Fuente: Esta investigación.

El usuario puede hacer doble clic sobre el mapa para ir agregando nodos a la geometría de la nueva ruta. Además si se hace clic sobre alguno de los nodos se despliega una ventana emergente con dos botones, los cuales permiten eliminar o crear un marcador entre dos marcadores.

Los datos ingresados en el formulario para la creación de rutas son enviados al controlador CreateRoutesControIIer.java, clase que gestiona la creación de rutas. Dentro de esta clase se encuentra un método denominado createRoutes() ,el cual crea y almacena la nueva ruta en la base de datos. A continuación se muestra un fragmento del código encargado de crear rutas.

```
        try {
            route.setName(route.getName().toUpperCase());
            route.setDetailRoute(route.getDetailRoute().toUpperCase());
            EJBRoute.create(route);
            this.listRoutesController.setRoutes(EJBRoute.findAll());

            messageController.infoMessage("Ruta creada");
        } catch (Exception e) {
            messageController.errorMessage("Ruta NO creada");
        }
    }
```

Para la visualización e interacción del usuario con el mapa se utilizó JavaScript. A continuación se muestra un fragmento de código.

```
map.on('dblclick', function (e) {
    latlong = e.latlng;
    if (geometry.length === 0) {
        geometry.push(latlong);
        polyline = L.polyline(geometry, {
            color: 'blue',
            opacity: 0.5,
            weight: 8
        }).addTo(map);
        drawMarker(latlong, false);
    }
```

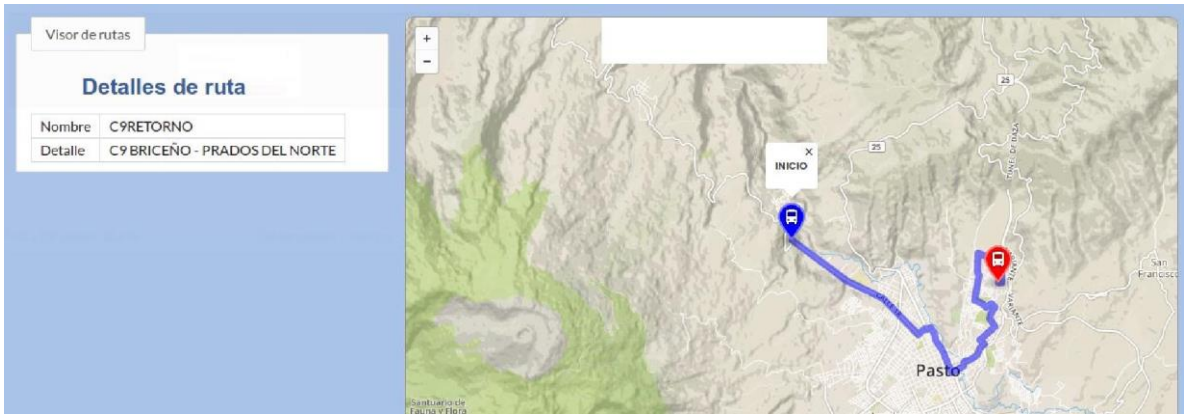
2.8. Ver y editar

En esta parte el usuario puede visualizar la información de una ruta determinada o actualizar su información.

Para la construcción de esta vista se hizo uso de un fieldSet, un dataTable y un commandButton.

- VER: Permite visualizar la información de la ruta seleccionada por el usuario, se puede observar los detalles así como también el recorrido que realiza, ver figura 13.

Figura 13. Información de ruta C9 RETORNO.



Fuente: Esta investigación.

Esta vista contiene elementos de primefaces tales como un fieldSet, cuatro outputLabel, un hl, elemento de HTML, y un mapa de MapBox.

La selección del usuario enviada al controlador, ViewRoutesController.java, clase encargada de enviar la información de la ruta seleccionada hacia el navegador del usuario. Para mostrar la información de la ruta seleccionada se hizo uso de JavaScript. A continuación se muestra un fragmento del código que permite visualizar la información de la ruta seleccionada.

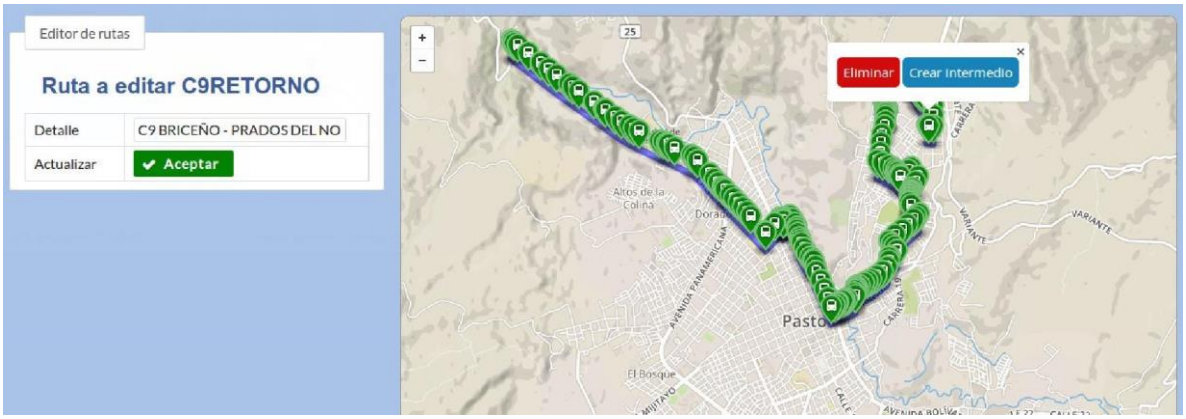
- Drawing polyline

```
/**
 * Drawing polyline
 */
  for (var i = 0; i < coordinate.length - 1; i = i + 2) {
     var latlon = L.latLng(coordinate[i], coordinate[i + 1]);
     points.push(latlon);
  }

  var polyline = L.polyline(points, {
     color: 'blue',
     opacity: 0.5,
     weight: 8
}).addTo(map);
```

- EDITAR: Esta opción permite actualizar la información de una determinada ruta del SET P. En la vista se muestran los detalles de ruta y además el recorrido que esta realiza. El usuario puede mover cada uno de los marcadores hacia la posición deseada y además al hacer clic sobre uno de ellos se muestra una ventana emergente que permite eliminar un marcador o crear uno intermedio, ver figura 14.

Figura 14. Editar ruta C9 RETORNO.



Fuente: Esta investigación.

La selección es enviada al controlador EditRoutesController.java, clase encargada de gestionar estas peticiones. Dentro de esta clase existe un método llamada

editRoutes(), el cual se encarga de la actualización de rutas. A continuación se muestra un fragmento de código que se encarga de permitir la edición de ruta.

```
try {

   route.setName(route.getName().toUpperCase());
   route.setDetailRoute(route.getDetailRoute().toUpperCase());
   EJBRoute.edit(route);
   this.listRoutesController.setRoutes(EJBRoute.findAll());

   messageController.infoMessage("Ruta actualizada");

} catch (Exception e) {
   messageController.errorMessage("Ruta NO actualizada");
}
```

Para mostrar el recorrido de ruta y además permitir el movimiento de los marcadores se hizo uso de JavaScript. A continuación se muestra un fragmento de código encargado de realizar esta tarea.

```
        /**
         * new marker has several events
         */
        auxMarker.on('mousedown', function (e) {
            latlong = e.latlng;
        });
        auxMarker.on('mouseup', function (e) {
            latLongFinal = e.latlng;
        });
        auxMarker.on('click', function (e) {
            latlong = e.latlng;
            $('#del').click(function () {
                deleteMarker(latlong);
            });
            $('#add').click(function () {
                createMarkerIntermediate(latlong);
            });
        });
        auxMarker.on('dragend', editRoute);

        /**
         * calculate distance between two markers
         */
        var distanceToLastMarker = parseInt(lstMarkers[lstMarkers.length -
1].getLatLng().distanceTo(auxMarker.getLatLng()).toFixed(0));
        var distanceToFirstMarker =
parseInt(lstMarkers[0].getLatLng().distanceTo(e.latlng).toFixed(0));

        if (distanceToLastMarker < distanceToFirstMarker) {
            lstMarkers.push(auxMarker);
            updatePolyine();
```

```
    } else {
        lstMarkers.unshift(auxMarker);
        updatePolyine();
    }
});

polyline = L.polyline(points, {
    color: 'blue',
    opacity: 0.5,
    weight: 8
}).addTo(map);

drawMarkers(points);
addEventMarkers();
```

El usuario puede hacer doble clic sobre el mapa para agregar nuevos puntos a la geometría.

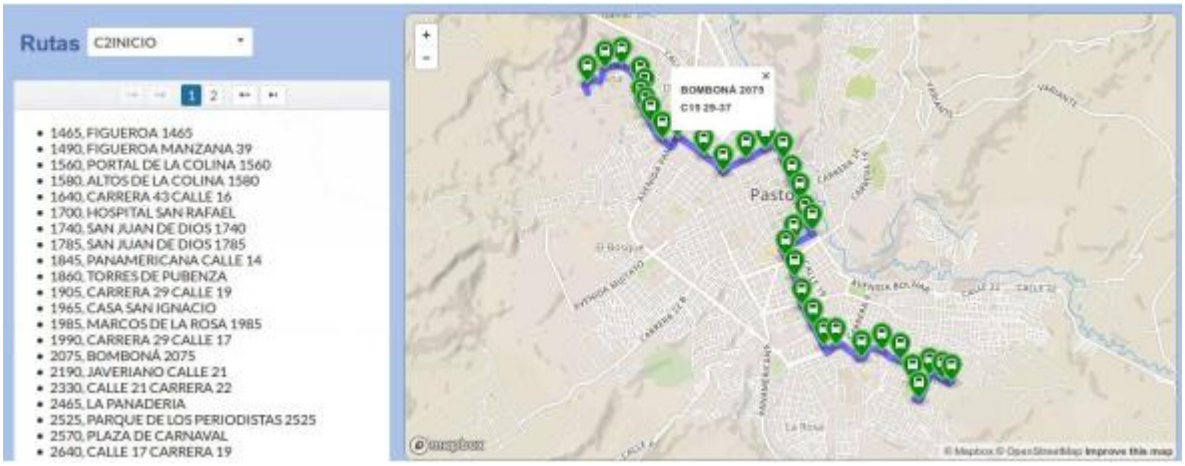
2.9. Paraderos asociados

Esta opción permite ver los paraderos asociados a una determinada ruta. El usuario puede ver un listado de todos los paraderos además se pinta sobre el mapa el recorrido de la ruta y los diferentes paraderos asociados a la ruta seleccionada, ver figura 15.

Esta vista tiene un hl, elemento de HTML, un selectOneMenu, un dataList, elementos de primefaces, y un mapa de MapBox.

El usuario puede hacer clic sobre alguno de los marcadores y se desplegará una ventana emergente con información acerca de la ubicación del paradero.

Figura 15. Paraderos asociados ruta C2 INICIO.



Fuente: Esta investigación.

La información de la ruta seleccionada es enviada al controlador StopsByRoutesControIler.java, esta clase es la encargada de consultar en la base de datos los paraderos asociados a una determinada ruta del SETP. Dentro de esta clase se encuentra un método llamado findStopsByRouteName() el cual consulta en la base de datos los paraderos asociados a la ruta seleccionada. A continuación se muestra un fragmento de código utilizado para lograr mostrar los paraderos asociados a una ruta.

```
try {
stops = EJBStopRoutes.stopsByRoutes(route.getName());
geometry = route.getGeometry();
if (stops.size() > 0) {
   for (Stops stop : stops) {
      dataMarkers += stop.getLatitude() + ";";
      dataMarkers += stop.getLongitude() + ";";
      dataMarkers += stop.getName() + ";";
      dataMarkers += stop.getLocalization() + ";";
   }
} else {
   messageController.errorMessage("RUTA SIN PARADEROS");
}
```

Del mismo modo que en funciones anteriores, en este caso también se utilizó JavaScript para mostrar los resultados en el navegador. A continuación se muestra un fragmento de código utilizado para mostrar los resultados de la selección en el navegador.

```
    /**
     * draw markers on the map
     */
    for (var i = 0; i < dataMarkers.length - 1; i = i + 4) {
        var latlng = L.latLng(dataMarkers[i], dataMarkers[i + 1]);
        marker = L.marker(latlng, {
            icon: L.mapbox.marker.icon({
                'marker-symbol': 'bus',
                'marker-color': '#088A08'
            })
        }).bindPopup("<strong>" + dataMarkers[i + 2] + " <br>  " + dataMarkers[i + 3]
+ "</strong>")
                .addTo(map);
        markers.push(marker);
    }

    /*
     * draw polyline
     */
    for (var i = 0; i < geometry.length - 1; i = i + 2) {
        var latlon = L.latLng(geometry[i], geometry[i + 1]);
        points.push(latlon);
    }
```

```
polyline = L.polyline(points, {
    color: 'blue',
    opacity: 0.5,
    weight: 8
});
```

MANUAL DE USUARIO

Módulo Administrativo

Grupo de Investigación Aplicada en Sistemas

Tabla de Contenido

Tabla de Figuras

MANUAL DE USUARIO FINAL

Título del software: Módulo administrativo SITApp.

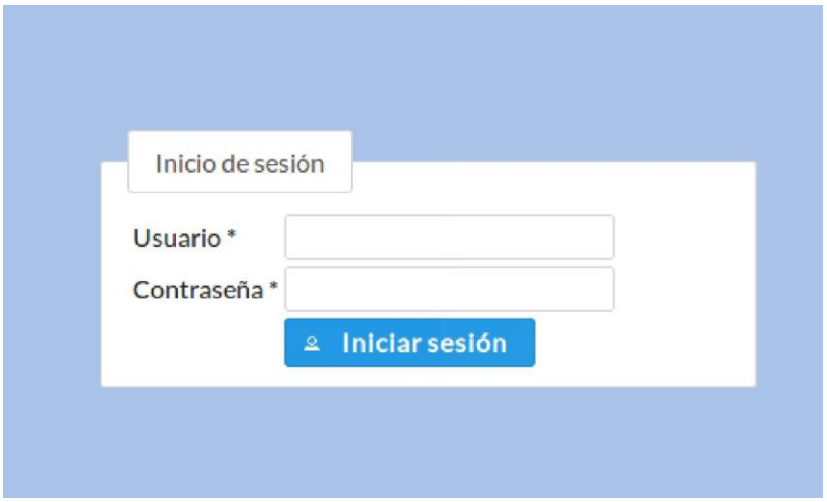
Documentación de desarrollo

La siguiente es la documentación que pretende guiar a los usuarios finales en el uso del aplicativo desarrollado. Cabe destacar que solo existe un rol asociado al sistema, administrador.

1. INICIO DE SESIÓN

La siguiente imagen, figura 1, es la presentación inicial del módulo administrativo de SITApp. El usuario, administrador, debe iniciar sesión para realizar cambios sobre la bases de datos del aplicativo móvil.

Figura 1. Interfaz inicial del módulo administrativo



Fuente: Esta investigación, aplicación en ejecución.

2. INTERFAZ SECUNDARIA DE LA APLICACIÓN

Una vez el usuario ha iniciado sesión, se muestra la interfaz, figura 2, mediante la cual puede realizar labores de actualización de rutas y paraderos del Sistema Estratégico de Transporte del municipio de Pasto (SETP).

Figura 2 . Interfaz secundaria del módulo administrativo.

Fuente: Esta investigación, aplicación en ejecución.

Esta interfaz contiene una barra de menú con 3 opciones, paraderos, rutas e inicio; estas opciones permiten gestionar la información de paraderos y rutas del SET P de la ciudad de Pasto. Además, la barra contiene dos botones de color rojo y verde desde los cuales el usuario puede cerrar sesión o cambiar la contraseña del usuario administrador. Los puntos, que se pueden ver sobre el mapa, son algunos paraderos del SETP del municipio de Pasto. Las opciones de paraderos y rutas se describen a continuación.

3. OPCIÓN PARADEROS

El usuario puede realizar las siguientes acciones:

- Eliminar paradero.
- Crear paradero.
- Ver y Editar.

- Asociar paraderos.
- Des-asociar paraderos.

3.1. Eliminar paradero

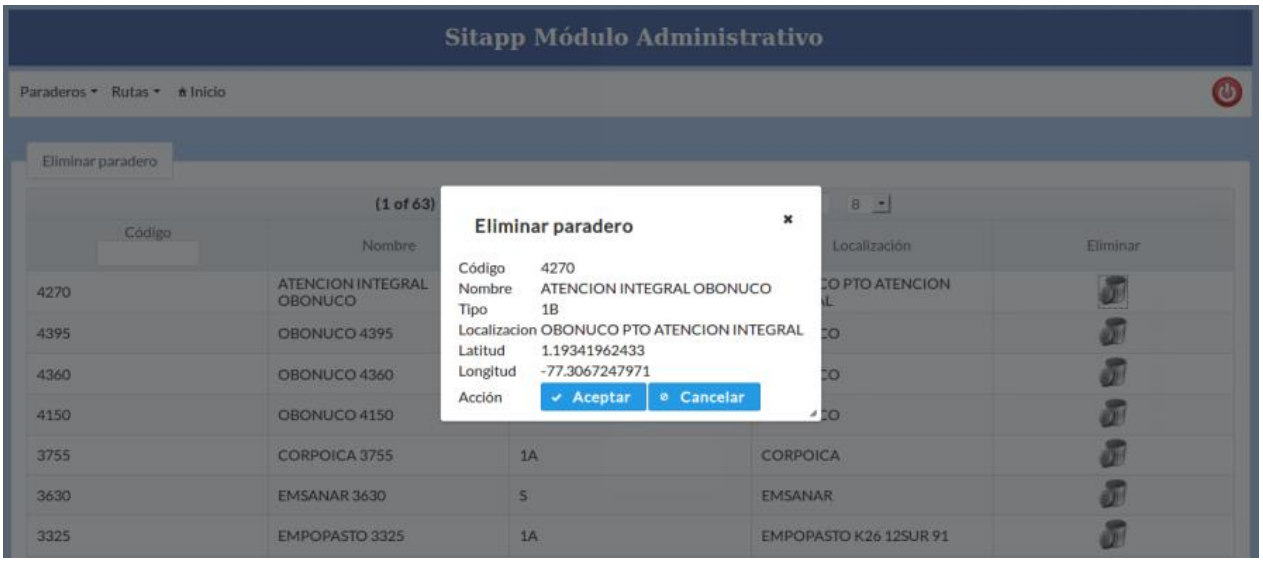
En esta opción el sistema muestra un listado de todos los paraderos que existen en la base de datos. El usuario puede buscar y seleccionar el paradero que desea eliminar, ver figura 3.

Figura 3. Listado de paraderos con la opción de eliminar.

Fuente: Esta investigación, aplicación en ejecución.

Para localizar el paradero a eliminar, el usuario tiene dos maneras para lograrlo. Escribir el código del paradero (número 1) o desplazarse haciendo uso de los números ubicados sobre la tabla de paraderos (número 2), una vez localizado el paradero el usuario debe presionar el botón eliminar (número 3). Cuando el usuario presiona el botón eliminar el sistema muestra una ventana emergente con los datos del paradero a eliminar y las opciones Aceptar o Cancelar, ver figura 4.

Figura 4. Ventana emergente con información de paradero seleccionado para ser eliminado.



Fuente: Esta investigación, aplicación en ejecución.

NOTA: Esta opción borra el paradero de la base de datos, si una o más rutas tienen asociado este paradero al presionar el botón aceptar la(s) ruta(s) quedara(n) sin este paradero dado que dejara de existir.

3.2. Crear paradero

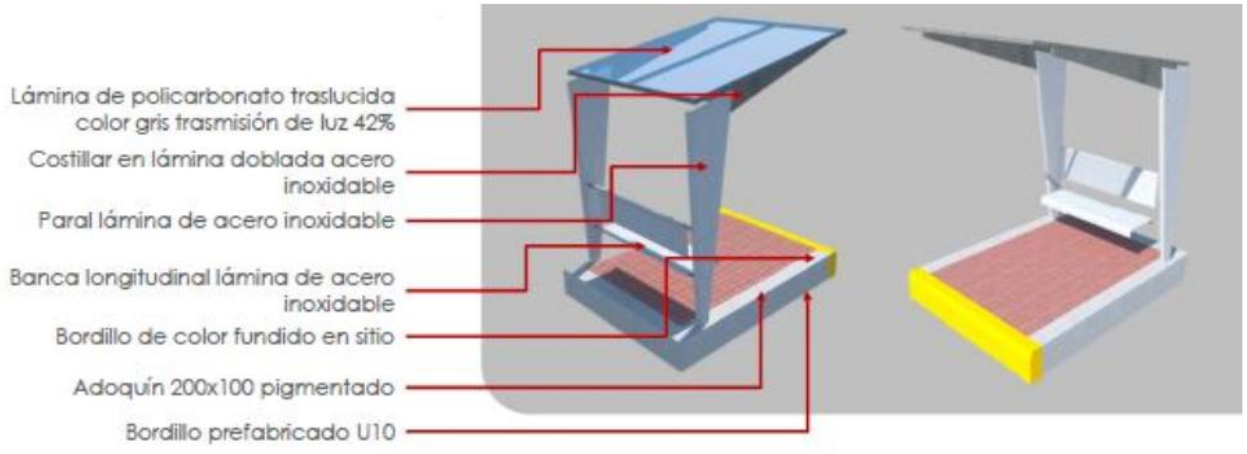
Esta opción permite al usuario administrador crear nuevos paraderos en el SET P. El usuario debe llenar datos referentes al paradero como son código, nombre, tipo, localización y además debe ubicar el paradero sobre un mapa, ver figura 5.

Figura 5. Crear paradero en el SETP.

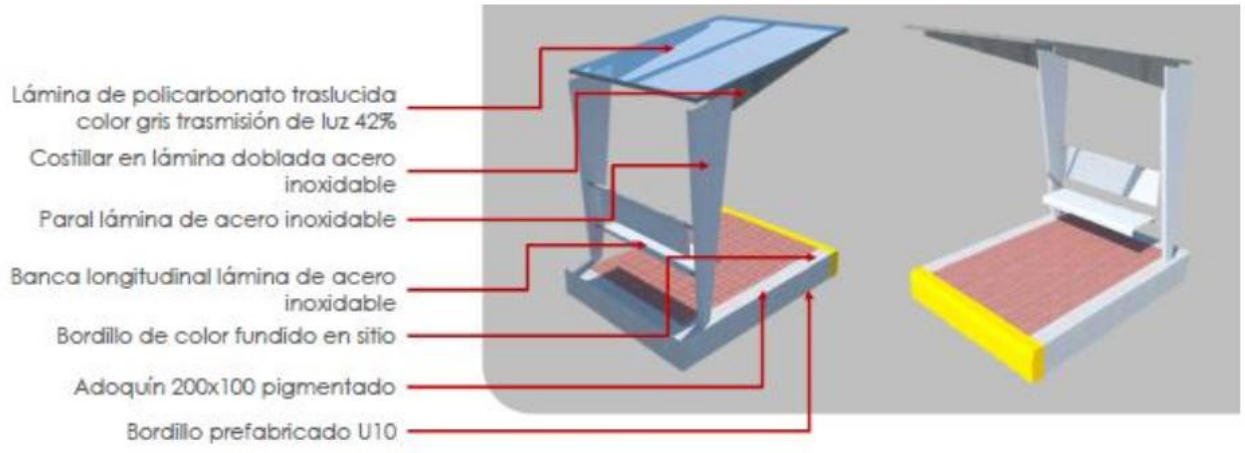
Fuente: Esta investigación, aplicación en ejecución.

Existen 11 tipos diferentes de paraderos pero estos son el resultado de una combinación de módulos así que para lograr una mayor comprensión de los tipos de paraderos que tiene el SETP de Pasto empezaremos por describir, a grandes rasgos, cada uno de los módulos.

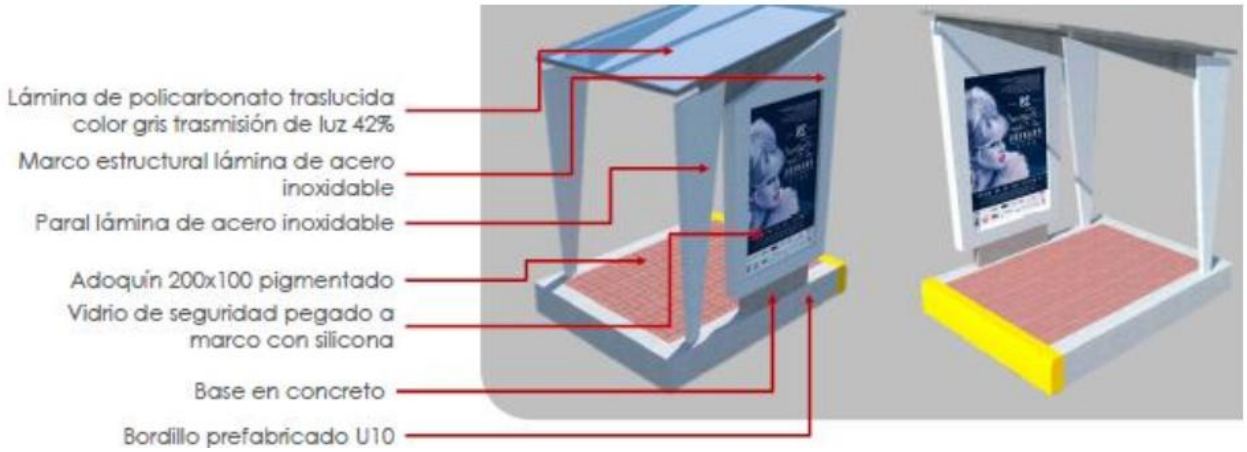
Módulo 1 (Ml)



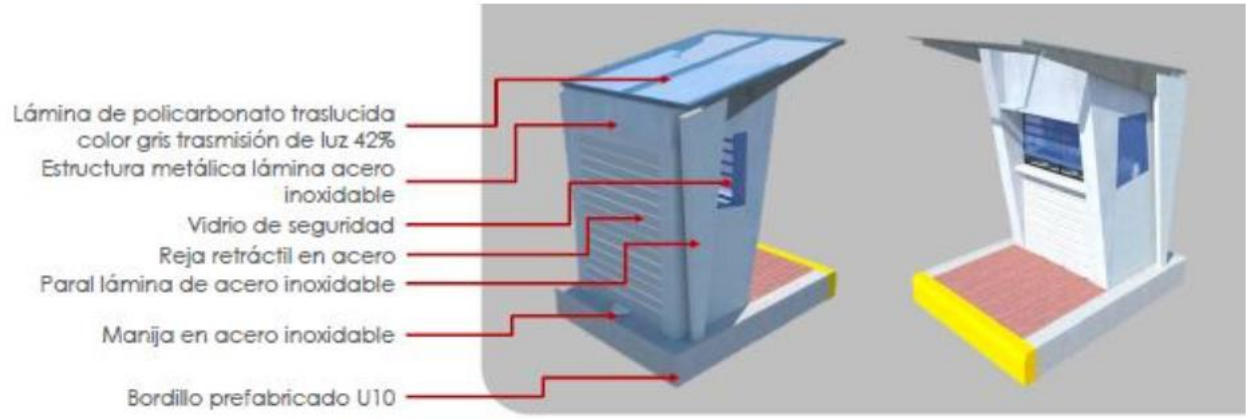
Módulo 2 (M2)



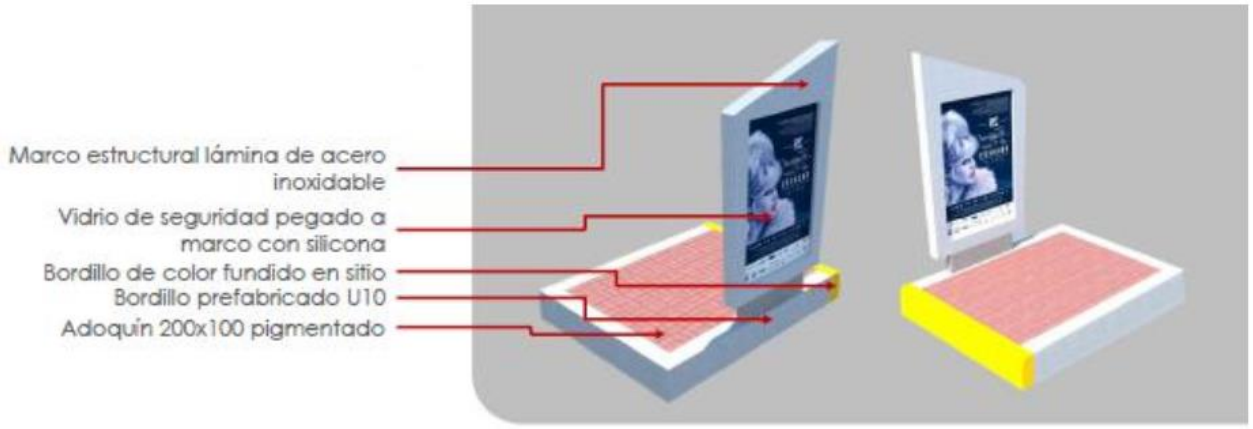
Módulo 3 (M3)



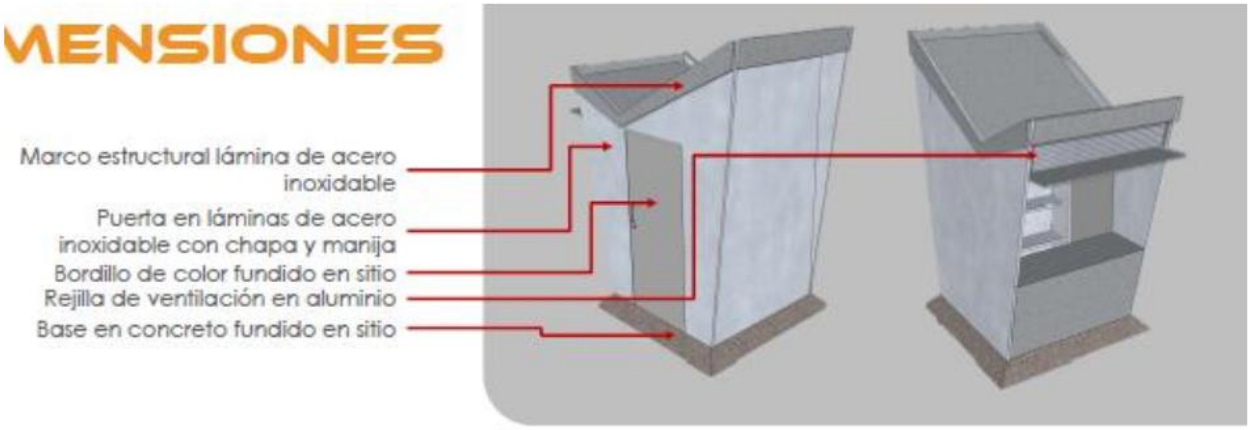
Módulo 4 (M4)



Módulo 5 (M5)



Módulo 6 (M6)



Ahora bien, los paraderos resultan de las combinaciones de los módulos anteriores, por lo tanto:

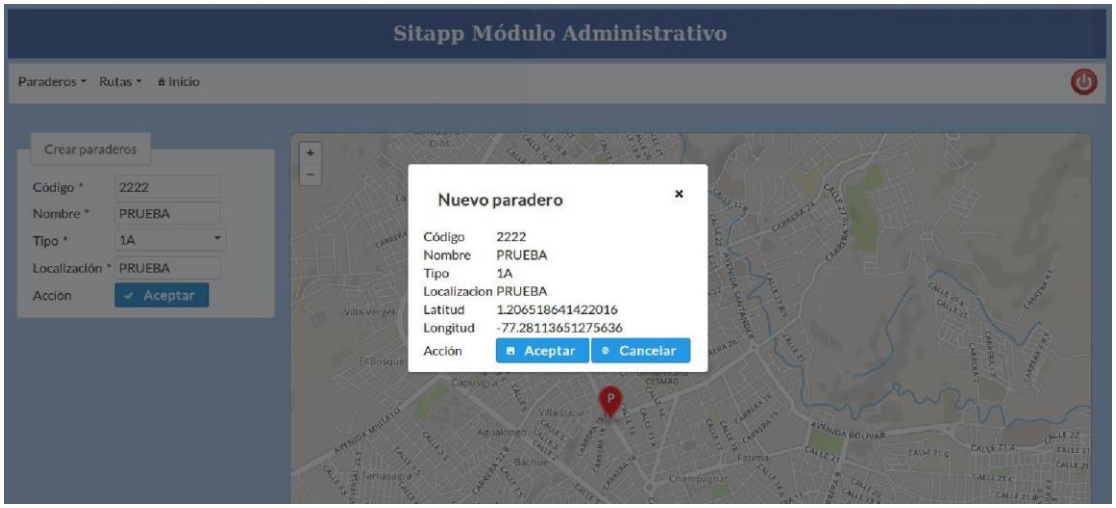
- Paradero PIA compuesto por módulos (Ml + Ml)
- Paradero PIB compuesto por módulos (M2 + M3)
- Paradero P2A compuesto por módulos (Ml + M2 + M3)
- Paradero P2B compuesto por módulos (M2 + M2 + M3)
- Paradero P2C compuesto por módulos (M2 + Ml + M3)
- Paradero P3A compuesto por módulos (M2 + Ml + M4 + M5)
- Paradero P3B compuesto por módulos (M2 + M2 + M4 + M5)
- Paradero P3C compuesto por módulos (Ml+ Ml + M4 + M5)
- Paradero 2+3 PENDIENTE

- Paradero S PENDIENTE
- Paradero NA PENDIENTE

NOTA: El usuario debe hacer doble clic sobre el mapa para que se dibuje el marcador y en el caso de que la posición no sea correcta puede arrastrar el marcador hasta la ubicación deseada. Se recomienda que el paradero este lo más cerca posible a la malla vial dado que al momento de asociar un paradero a una ruta debe existir una distancia mínima entre el paradero y la vía, así que si el paradero está demasiado lejos no será posible asociarlo a la ruta deseada.

Al presionar el botón aceptar se muestra una ventana emergente con la información del nuevo paradero y las opciones Aceptar o Cancelar, ver figura 6.

Figura 6. Ventana emergente con los datos del nuevo paradero.

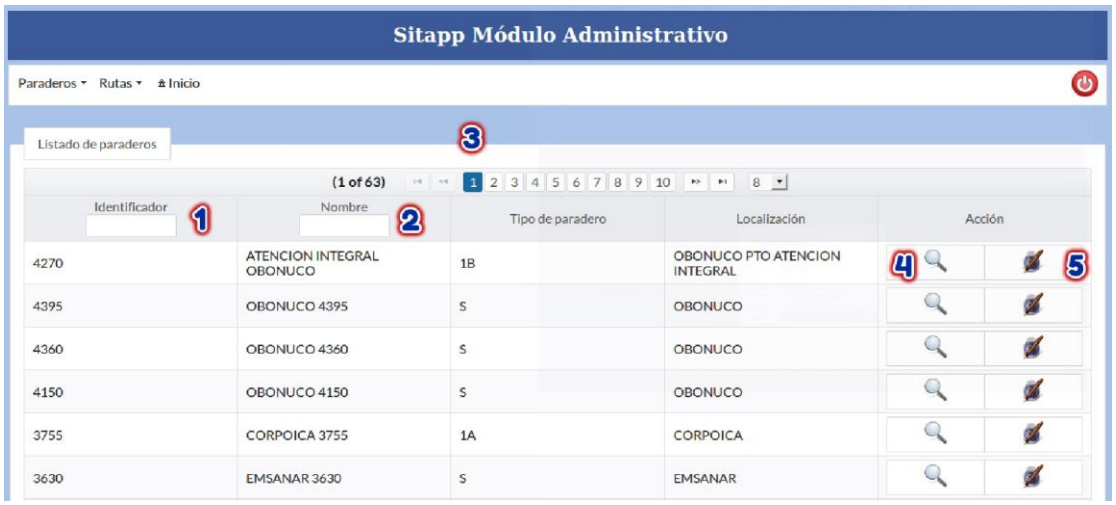


Fuente: Esta investigación, aplicación en ejecución.

3.3. Ver y editar

Esta opción presenta un listado de paraderos y dos botones con la opción Ver o Editar, ver figura 7.

Figura 7. Listado de paraderos con las opciones Ver y Editar.



Fuente: Esta investigación, aplicación en ejecución.

Al igual que en la opción de Eliminar, el usuario tiene varias formas para ubicar el paradero deseado. El usuario puede escribir el código del paradero (número 1), nombre del paradero (número 2) o utilizar los números ubicados en la parte superior del listado de paraderos (número 3). Las opciones Ver y Editar están en los botones con forma de lupa (número 4) y tintero (número 5) respectivamente.

3.3.1. Ver paradero

La opción Ver, re-direcciona a una página la cual muestra información y ubicación en el mapa del paradero seleccionado, ver figura 8.

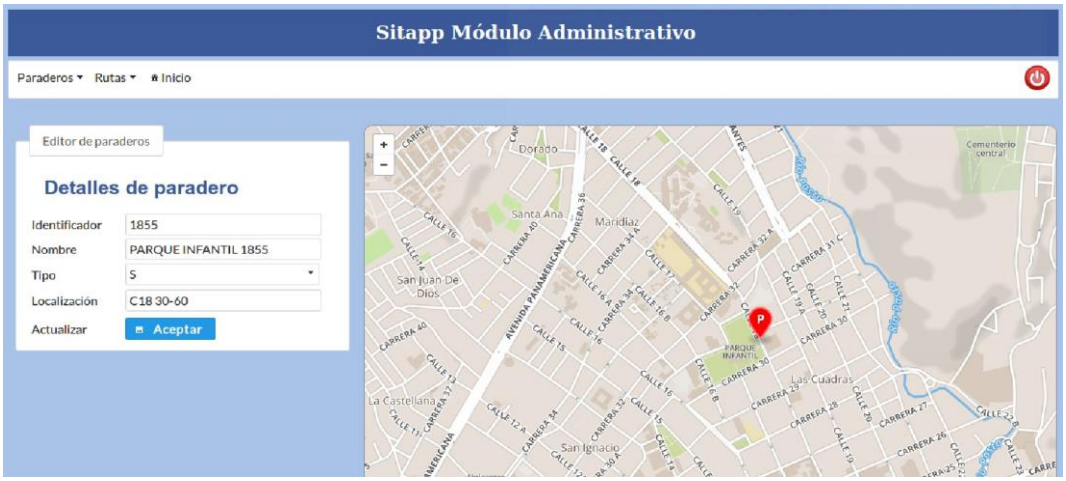
Figura 8. Página con información del paradero seleccionado.

Fuente: Esta investigación, aplicación en ejecución.

3.3.2. Editar paradero

La opción Editar, muestra la información y ubicación del paradero seleccionado, pero permite editarla, ver figura 9.

Figura 9. Información y ubicación de paradero seleccionado.



Fuente: Esta investigación, aplicación en ejecución.

NOTA: Para actualizar la ubicación del paradero sencillamente se arrastra el marcador hacia la posición deseada y una vez realizados los cambios se presiona el botón Aceptar.

3.4. Asociar paraderos

Esta opción permite asociar nuevos paraderos a las rutas del SET P, ver figura 10. El usuario debe seleccionar la ruta, el sistema busca automáticamente todos los paraderos que no están asociados a esa ruta y los muestra en la sección paraderos donde se debe seleccionar el paradero que se desea asociar a la ruta.

Consideraciones, El paradero no puede estar a más de 2 metros de la ruta. La secuencia, hace referencia al orden de los paraderos, puede estar entre el primer paradero y el último + 1, para los demás casos generará un error.

NOTA: El aplicativo móvil trabajo con los paraderos y estos son los que dan el sentido de la ruta, si va o vuelve, esto debido a que no existe una conexión con los GPS de los buses, es por eso que se debe tener cuidado al momento de asociar un paradero, CORROBORAR QUE EL ORDEN SEA EL CORRECTO. Para corroborar el orden se puede hacer clic sobre los marcadores para ver información relacionada.

Figura 10. Asociar paraderos a una determinada ruta del SETP

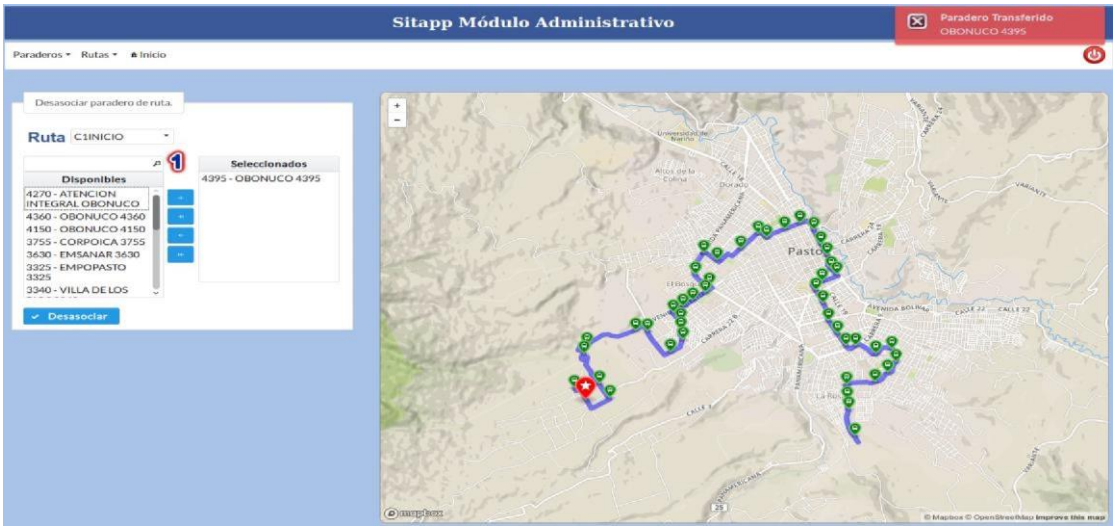
Fuente: Esta investigación, aplicación en ejecución.

3.5. Desasociar paraderos

Esta opción permite eliminar paraderos asociados a una ruta pero no los borra de la base de datos, únicamente borra el enlace entre la ruta y el paradero pero este último sigue almacenado en la base de datos.

El usuario debe seleccionar la ruta y el sistema lista automáticamente los paraderos asociados a esa ruta en específico y los ubica sobre un mapa, ver figura 1 1, hacer doble clic sobre el paradero a des-asociar, el cual será enviado al otro lado de la lista, y presionar el botón desasociar el sistema eliminará el enlace que existe entre la ruta y el paradero seleccionado.

Figura 11. Desasociar paraderos de ruta.



Fuente: Esta investigación, aplicación en ejecución.

Para ubicar el paradero a desasociar el usuario tiene dos maneras de hacerlo, escribir el código del paradero (número 1) o desplazarse hacia abajo hasta encontrar el paradero deseado.

4. OPCIÓN RUTAS

El usuario tiene las siguientes opciones:

- Eliminar ruta.
- Crear ruta.

- Ver y Editar.
- Paraderos asociados.

4.1. Eliminar ruta

El sistema despliega un listado de las rutas almacenadas en la base de datos y las muestra en una tabla, ver figura 12, además de la información de las rutas, también se tiene un botón, con forma de cesto de basura (número 3), el cual permite eliminar la ruta seleccionada.

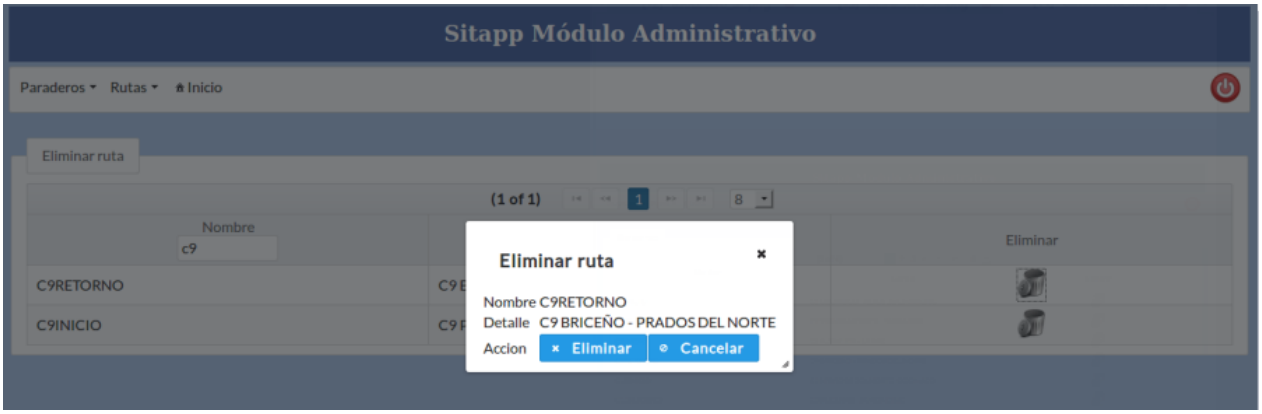
Figura 12. Listado de rutas con la opción de eliminar.

Fuente: Esta investigación, aplicación en ejecución.

Para localizar la ruta a eliminar el usuario tiene dos maneras para lograrlo. Haciendo uso del cuadro de búsqueda (número 1) ubicado sobre el nombre de la ruta o utilizando los números ubicados sobre el listado de rutas (número 2), una vez localizada la ruta a eliminar se debe presionar el botón con forma de cesto de basura (número 3).

Al momento de presionar el botón eliminar, se despliega una ventana emergente con la información de la ruta a eliminar y dos botones, Aceptar o Cancelar, ver figura 13.

Figura 13. Ventana emergente con información de la ruta seleccionada para ser eliminada.



Fuente: Esta investigación, aplicación en ejecución.

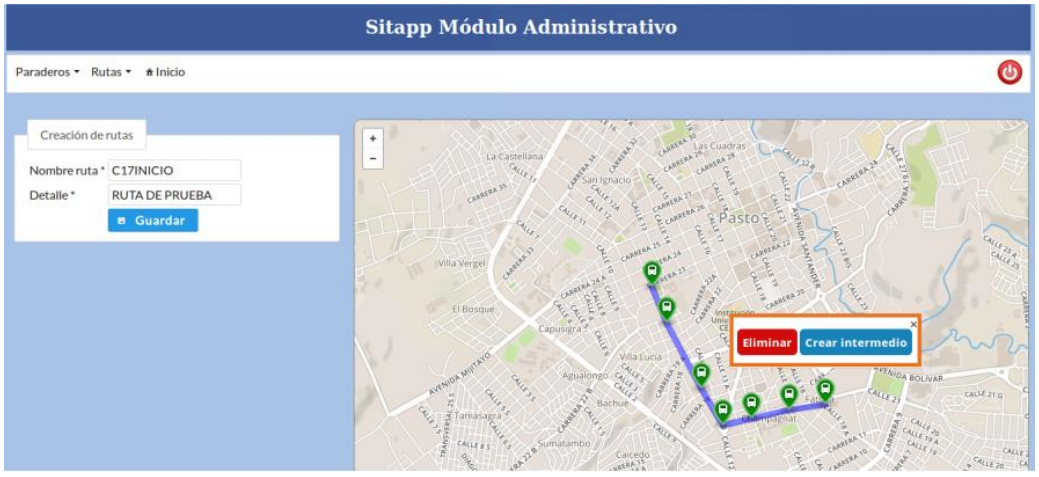
NOTA: Esta opción elimina la ruta de la base de datos, se debe proceder con cuidado.

4.2. Crear ruta

Esta opción permite crear rutas en el SET P de la ciudad de Pasto.

El sistema muestra un formulario dentro del cual se debe llenar información de la ruta a crear, la información es obligatoria, ver figura 14.

Figura 14. Formulario de creación de rutas.



Fuente: Esta investigación, aplicación en ejecución.

Para crear la geometría de la ruta, el usuario debe hacer doble clic sobre el mapa para que el sistema cree un nuevo nodo y esta operación se debe repetir hasta lograr el resultado requerido.

Al hacer clic sobre alguno de los nodos se despliega una ventana emergente. Esta ventana permite eliminar el nodo o crear un nodo intermedio entre el anterior y el siguiente.

Los nodos se pueden dibujar al inicio o fin de la geometría, basta con hacer doble clic y el sistema determinará si lo añade al final o al inicio. Con esto logramos que la ruta pueda ser extendida al inicio o al final.

Cuando se ha logrado el resultado requerido se presiona el botón guardar y el sistema almacena la ruta en la base de datos. Luego se puede asociar los diferentes paraderos que pertenecen a la ruta que se acaba de crear.

4.3. Ver y editar

En esta opción, el sistema muestra un listado de las rutas existentes en la base de datos. El listado además de contener información referente a las rutas tiene dos botones, uno con forma de lupa y otro con forma de tintero, estos botones permiten Ver y/o Editar una ruta determinada, ver figura 15.

Figura 15. Lista de rutas con las opciones Ver y Editar.

Fuente: Esta investigación, aplicación en ejecución.

Para ubicar la ruta que se desea editar el usuario tiene tres opciones, escribir el nombre de ruta (número 1), escribir una palabra clave en el cuadro detalle (número 2) o utilizar los números ubicados sobre el listado de rutas. Una vez localizada la ruta se tiene dos opciones, Ver (número 4) o Editar (número 5).

4.3.1. Ver ruta

Al presionar el botón ver, se re-direcciona a una nueva página donde se muestra información de la ruta, nombre, detalle y se pinta sobre un mapa, ver figura 16.

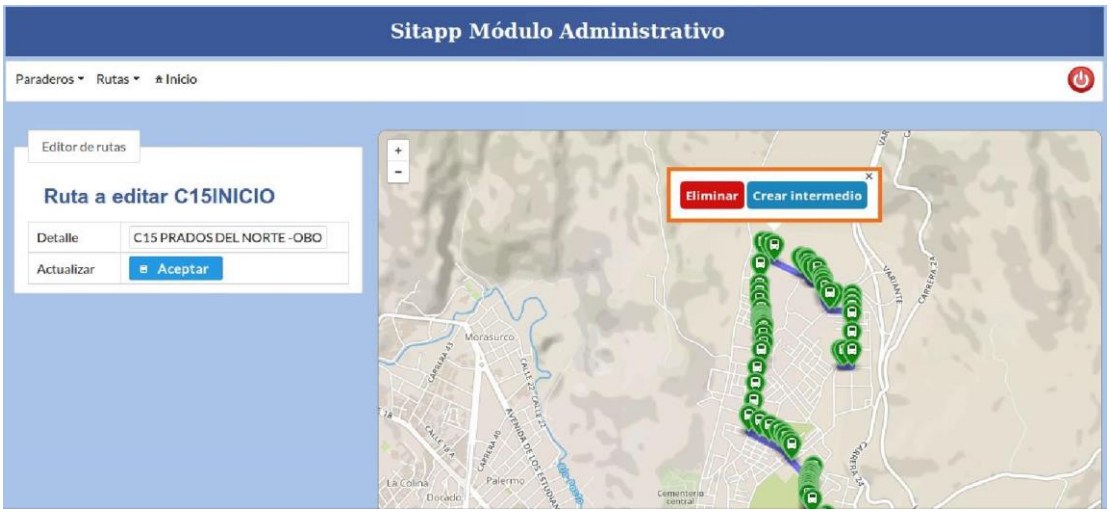
Figura 16. Información y recorrido de ruta seleccionada.

Fuente: Esta investigación, aplicación en ejecución.

4.3.2. Editar ruta

Al presionar el botón Editar, se re direcciona a una nueva página, pero esta permite actualizar la información de la ruta además de su recorrido, ver figura 17.

Figura 17. Editar ruta seleccionada.



Fuente: Esta investigación, aplicación en ejecución.

El usuario puede editar la información referente al detalle de ruta y el recorrido de la misma. Para editar el recorrido de la ruta, se puede lograr arrastrando los diferentes nodos que se han creado hasta la posición deseada, además, al hacer clic sobre alguno de los nodos se muestra una ventana emergente con dos botones al interior los cuales permiten eliminar o crear un nodo intermedio.

La ruta puede ser extendida hacia el inicio o al final, basta con hacer doble clic y el sistema determinará si lo adiciona al final o al inicio de la geometría.

4.4. Paraderos asociados

Esta opción permite ver los paraderos asociados a una determinada ruta, ver figura 18. El usuario selecciona la ruta y el sistema muestra un listado de los paraderos que tiene asociados y además dibuja el recorrido de la ruta con cada uno de los paraderos sobre un mapa.

Figura 18. Paraderos asociados a ruta seleccionada.

Fuente: Esta investigación, aplicación en ejecución.

Al hacer clic sobre alguno de los paraderos se presenta, en una ventana emergente, el nombre y la ubicación del paradero.