Universidad Nacional Autónoma de México

Facultad de Ingeniería

División de Ingeniería Eléctrica

Ingenieria en Computacion

Trabajo Final

Asignatura: Cómputo Móvil

Grupo: 03

Semestre 2026-1

Fecha de entrega: 28 de octubre de 2025

Profesor: Ing. Marduk Pérez de Lara Domínguez

Equipo: 3 (Lambda)

Integrantes:

- Gómez Vázquez Juan Pablo
- Martínez Miranda Juan Carlos
- Suaznavar Arvizu Oscar Manuel
- Uriarte Ortiz Enrique Yahir

Requerimientos de la Aplicación *Conecta Fácil*

Conecta Fácil está enfocada en brindar rutas accesibles para usuarios con movilidad reducida, así como un módulo de emergencia (SOS). A continuación, se describen los requerimientos del sistema, divididos en reglas de negocio, requerimientos funcionales y requerimientos no funcionales.

- Reglas de negocio: Definen las políticas y restricciones bajo las cuales opera la app. Por ejemplo:
- *Conecta Fácil* ofrecerá únicamente rutas que cumplan criterios de accesibilidad (evitando escaleras u obstáculos para sillas de ruedas).
- La funcionalidad SOS solo estará disponible para usuarios ubicados dentro del área de cobertura del servicio (p. ej., dentro de la ciudad soportada), enviando alertas únicamente a contactos de emergencia predefinidos.
- Los datos de ubicación del usuario se utilizarán exclusivamente para calcular rutas y activar el SOS, cumpliendo con políticas de privacidad (no se comparten con terceros sin consentimiento).

Requerimientos funcionales:

- Mostrar rutas accesibles: La app permitirá ingresar un destino y mostrará una ruta optimizada y accesible desde la ubicación actual del usuario, evitando barreras arquitectónicas conocidas.
- Búsqueda de ubicaciones: El usuario podrá buscar direcciones, lugares de interés o puntos accesibles mediante un buscador integrado. La aplicación mostrará sugerencias de lugares conforme se escribe.
- Visualización de mapas y navegación: Se integrará un mapa interactivo que mostrará la ruta trazada. El usuario podrá visualizar indicaciones paso a paso (giros, tramos con rampas, cruces peatonales, etc.).
- Módulo SOS: Disponibilidad de un botón de emergencia que al activarse enviará la ubicación actual y un mensaje de auxilio a un contacto de emergencia o servicio de asistencia. El proceso solicitará confirmación para evitar activaciones accidentales.
- Gestión de contacto de emergencia: El usuario podrá registrar o seleccionar un contacto de emergencia (nombre y número telefónico) dentro de la app, que será usado por el SOS.
- Pantalla de información y ayuda: Incluir una sección con instrucciones de uso de la app, información sobre accesibilidad y datos de contacto o soporte técnico.

Requerimientos no funcionales:

- Usabilidad y Accesibilidad: La aplicación debe ser intuitiva y usable por la mayoría de los usuarios, incluyendo personas con discapacidad. Se seguirán lineamientos de diseño inclusivo, como alto contraste de colores, botones

grandes y navegación coherente. La app será compatible con VoiceOver (lector de pantalla en iOS) y otras tecnologías de asistencia.

- Rendimiento: El cálculo de rutas y la carga del mapa deben ocurrir en pocos segundos (idealmente <2 s para respuestas de interfaz), proporcionando una experiencia fluida. Incluso en dispositivos de gama media, la app debe rendir adecuadamente.
- Conectividad y offline: *Conecta Fácil* requiere conexión a Internet para obtener datos de mapas y rutas. No obstante, deberá manejar con gracia la pérdida de conexión (por ejemplo, mostrar la última ruta cargada o un mensaje de "sin conexión" en lugar de fallar).
- Seguridad y Privacidad: La app solicitará únicamente los permisos necesarios (ubicación y opcionalmente micrófono para comandos de voz). Al pedir acceso a la ubicación por primera vez, mostrará la razón de uso de forma clara. Los datos personales (como contactos de emergencia) se almacenarán de forma segura (localmente en el dispositivo) y no serán compartidos externamente.
- Compatibilidad: La aplicación debe ser compatible con la versión de iOS vigente y al menos una o dos versiones anteriores (por ejemplo, iOS 16+), para abarcar un amplio número de dispositivos objetivo. Asimismo, debe adaptarse a distintos tamaños de pantalla y orientaciones (detallado más adelante).
- Mantenibilidad: El código fuente debe estructurarse de forma modular y documentada, facilitando futuras extensiones (p. ej., agregar más rutas personalizadas) y simplificando las pruebas y el mantenimiento.

Funcionalidades Propuestas

Durante la planificación de *Conecta Fácil* se evaluaron ciertas funcionalidades avanzadas para determinar su viabilidad técnica dentro del alcance del proyecto. En particular, se consideraron: rutas personalizadas para el usuario e integración con servicios de mapas externos. A continuación se justifica la viabilidad o inviabilidad de estas propuestas:

- Rutas personalizadas para el usuario: Esta funcionalidad implicaría adaptar y optimizar las rutas en función de preferencias o historial de cada usuario (por ejemplo, rutas favoritas, evitar pendientes empinadas, etc.). Tras el análisis, se determinó que implementar rutas altamente personalizadas excede las capacidades del MVP debido a la falta de datos suficientes y la complejidad algorítmica. Requeriría recopilar información del comportamiento del usuario a lo largo del tiempo y aplicar algoritmos de machine learning o reglas de negocio complejas para recomendar trayectos individuales. Técnicamente es posible, pero no viable en el corto plazo del proyecto sin una infraestructura robusta y bases de datos geoespaciales específicas. Por ende, en esta primera versión se optó por ofrecer rutas accesibles estándar para todos los usuarios, enfocadas en criterios generales de accesibilidad, y dejar la personalización avanzada como mejora futura. Esta decisión garantiza un desarrollo más sencillo y centrado en

la calidad de las funciones básicas, aplicando el principio de MVP (producto mínimo viable) para validar primero la funcionalidad principal.

- Integración con mapas y datos de localización: Se evaluó cómo implementar el sistema de mapas y cálculo de rutas. Dos caminos principales fueron considerados: utilizar herramientas nativas de iOS (MapKit y CoreLocation) o integrar un servicio externo como Google Maps API. Viabilidad: La integración de un mapa interactivo es técnicamente viable mediante *MapKit*, el framework nativo de Apple. MapKit permite mostrar mapas y obtener rutas (indicaciones) directamente dentro de la app, sin costo adicional por uso y con una fácil integración a Xcode. Dado que el requerimiento del cliente es un desarrollo nativo iOS, aprovechar las herramientas nativas resulta ideal. Adicionalmente, *CoreLocation* proporciona acceso al GPS del dispositivo para obtener la ubicación del usuario sin complicaciones. Por otro lado, la API de Google Maps ofrece datos muy ricos (por ejemplo, información de tráfico, más cobertura de POIs), pero su uso conlleva la obtención de una clave de API y posibles costos por volumen de solicitudes una vez superada la cuota gratuita. Considerando el alcance del proyecto académico (y que MapKit cubre las necesidades básicas de rutas), se decidió emplear MapKit. Esto hizo la integración con mapas más directa y mantuvo los costos en cero, a la vez que garantiza una experiencia consistente en iOS (aprovechando la interfaz y gestos nativos del mapa de Apple). No obstante, sí se identificaron limitaciones: ni MapKit ni Google Maps proveen actualmente opciones explícitas para *"rutas accesibles para silla de ruedas"* en recorridos a pie, por lo que inicialmente la aplicación marcará rutas peatonales estándar. Se justifica esta elección dado el tiempo de desarrollo disponible, mencionando al usuario que la ruta sugerida es una base y que la accesibilidad real puede variar. En versiones futuras, se podría considerar integrar datos abiertos de ciudades (OpenStreetMap u otros) que contengan etiquetas de accesibilidad en calles y aceras, o incluso usar la API de Directions de Google con "modo accesible" en transporte público donde exista esa opción. En resumen, técnicamente la integración de mapas es viable con las herramientas actuales, pero algunas funcionalidades avanzadas (como garantía de accesibilidad total en cada ruta o personalización por preferencias de usuario) quedan identificadas como no viables en MVP por requerir datos y desarrollo adicionales significativos.

Alcance del Proyecto y Definición del MVP

Alcance del Proyecto: El proyecto *Conecta Fácil* abarca el diseño, desarrollo y prueba de una aplicación móvil iOS cuyo propósito principal es conectar a los usuarios con rutas de transporte peatonal accesibles, proporcionando además una herramienta de asistencia de emergencia. El alcance incluye funcionalidades de búsqueda de rutas, visualización en mapa, indicaciones detalladas y envío de alertas SOS con ubicación. También comprende componentes de apoyo como pantallas de configuración (contacto de emergencia) e información. Quedan fuera de alcance en esta fase inicial varias

características propuestas en etapas tempranas, como la integración de múltiples perfiles de usuario, gamificación (ej. recompensas por rutas recorridas) o una base de datos colaborativa de puntos accesibles. Estas ideas se documentaron para posible desarrollo futuro, pero no se implementarán en el presente proyecto, para así concentrar recursos en las funciones esenciales y asegurar su calidad.

Definición del MVP: Se define el Producto Mínimo Viable (MVP) de *Conecta Fácil* como aquella versión funcional de la aplicación que entrega valor inmediato al usuario resolviendo el problema central: encontrar rutas seguras y accesibles, y pedir ayuda en caso necesario. En concreto, el MVP incluirá dos módulos fundamentales:

- Módulo de Rutas Accesibles: Permite al usuario ingresar un destino y obtener una ruta adecuada en el mapa, con indicaciones paso a paso. Esta ruta priorizará la accesibilidad (p. ej., evitar escaleras y obstáculos conocidos) y mostrará información relevante como la distancia y el tiempo estimado. Se considera exitoso el MVP si un usuario con movilidad reducida puede usar la app para planificar un trayecto más conveniente que uno genérico.

- Módulo SOS (Emergencia): Proporciona al usuario un medio rápido de contactar ayuda enviando su ubicación actual a un contacto de confianza o servicio de emergencias. En el MVP, esto se concretará mediante un botón *SOS* que, al ser confirmado, envía vía SMS los datos de localización (enlace a mapa) y un mensaje predefinido al contacto de emergencia configurado. Este módulo atiende la necesidad crítica de asistencia en ruta, dando tranquilidad al usuario. Un MVP exitoso garantiza que el mensaje SOS se entrega correctamente y que el proceso es sencillo (pocos toques) para el usuario en situación de estrés.

El alcance del MVP cubre entonces: búsqueda de lugares, cálculo de una ruta accesible con mapa e instrucciones, visualización interactiva de la ruta en el dispositivo, gestión básica de un contacto de emergencia, y envío de alerta SOS mediante funciones nativas del teléfono (llamada o SMS). Todo lo anterior operando dentro de un entorno iOS nativo. Con este MVP funcional –enfatizando *rutas accesibles* y *SOS*– se podrá obtener retroalimentación valiosa de los usuarios iniciales y validar la propuesta central antes de invertir en funcionalidades complementarias. Cabe destacar que un MVP así definido se alinea con las recomendaciones de desarrollo ágil: entregar un producto con suficientes características para satisfacer a los usuarios iniciales y obtener retroalimentación para el desarrollo futuro.

Análisis de Servicios y Datos por Pantalla

En esta sección se detalla, por cada pantalla o módulo, la naturaleza de los datos manejados, las operaciones CRUD (Crear, Leer, Actualizar, Borrar) involucradas, las conexiones con servicios o APIs externas, requerimientos de autenticación, formatos de datos y costos potenciales. También se indica el uso de almacenamiento local cuando aplique y la justificación del mismo.

Pantallas de Onboarding (1-3): Estas pantallas de introducción manejan únicamente datos estáticos locales (textos descriptivos e imágenes ilustrativas) que vienen integrados en la aplicación. Operaciones: Solo lectura (Read) – la app carga y muestra este contenido. No hay creación ni modificación de datos por parte del usuario, salvo registrar internamente que el onboarding fue mostrado (una bandera booleana en almacenamiento local, para no repetirlo). Servicios/APIs: No requieren conexión a ningún servicio externo, ya que todo el contenido está embebido. Autenticación: No aplica. Formato de datos: Texto plano y gráficos en formato PNG/SVG dentro de la app. Costo: Nulo, puesto que no hay consumo de APIs de terceros ni licencias. Almacenamiento local: Sí, se puede almacenar un valor como `onboardingVisto=true` (por ejemplo en NSUserDefaults) para saltar estas pantallas en próximos usos. Este dato se guarda localmente por conveniencia del usuario.

Pantalla Principal (Mapa y Búsqueda): Esta pantalla es una de las más dinámicas. Tipo de datos: Geoespaciales – coordenadas GPS del usuario, mapas y rutas. Operaciones:
- *Crear:* Internamente crea una petición de ruta cuando el usuario ingresa un destino (no es crear en BD, sino crear una solicitud a la API).
- *Leer:* – Ubicación actual: la app lee la ubicación usando el servicio de GPS del dispositivo (Core Location). – Mapa e imágenes de mapa: lee mapas de la API de MapKit (cargas de los tiles del mapa). – Ruta: lee el resultado de la consulta de indicaciones (lista de pasos, polílinea de ruta).
- *Actualizar:* actualiza en tiempo real la posición del usuario en el mapa si este se mueve; también actualiza la ruta mostrada si se recalcula.
- *Eliminar:* podría eliminar o limpiar una ruta previa del mapa al pedir una nueva, pero esto es manejo en memoria temporal, no un borrado persistente.

Conexión con servicios/APIs: Sí, aquí la app se conecta con los servicios de Apple (MapKit/Mapas) para descargar mapas y calcular la ruta. Mediante *MKDirections* se envía origen (ubicación actual) y destino (seleccionado) y se recibe un objeto ruta. Esto requiere Internet activo. Si se hubiera usado Google Maps API, se realizaría una solicitud HTTPS al endpoint de Directions de Google con los parámetros, recibiendo un JSON con la ruta; en tal caso habría que parsear ese JSON manualmente. En este proyecto, usando MapKit, el manejo es más transparente (MapKit entrega directamente objetos nativos).
Autenticación: No se necesita autenticación de usuario para usar MapKit. Sin embargo, si fuera Google Maps API, se requeriría incluir una *API Key* en la app para autenticarse con Google (lo cual tiene consideraciones de seguridad para no exponerla públicamente). Con MapKit evitamos esta complicación (la autenticación está implícita en la cuenta de desarrollador de Apple y no requiere credenciales en la app).
Formatos de datos: – *Coordenadas:* numéricas (latitud, longitud en dobles). – *Direcciones:* texto (por ejemplo, “Calle X #123”). – *Ruta:* objeto complejo en memoria (en MapKit), o JSON si fuera externo. Con MapKit podríamos obtener las indicaciones como texto o utilizar la clase `MKRoute` que incluye polilíneas. En caso de JSON externo: vendría en formato JSON con pasos y coordenadas codificadas (como polylines encoded).
Costos: Usar MapKit es gratuito para un volumen típico de usuarios (Apple no cobra

por solicitudes de mapa/ruta en apps iOS). Si hubiéramos usado Google Directions API, los primeros 100,000 pedidos mensuales son gratuitos; más allá, se pagaría por cada consulta. Este costo no se asume en MVP al optar por MapKit.
Almacenamiento local: Mínimo. Se podría cachear la última ruta consultada (por ejemplo, guardar el destino último para autocompletarlo si la app se cierra y abre), pero no es crítico. No se almacena de forma persistente la ruta entera. Solo se almacena localmente la *ubicación actual* si el usuario da permiso para usarla en segundo plano (no implementado en MVP), pero en nuestro caso la ubicación se usa en tiempo real y no se registra permanentemente. La razón de no almacenar rutas es que son altamente contextuales (válidas en ese momento) y sería engañoso mostrarlas después, dado que las condiciones pueden cambiar.

Pantalla de Resultados de Búsqueda (Destinos):
Tipo de datos: Lista de sugerencias de lugares (nombre, dirección, coordenadas posiblemente). Operaciones:
- *Crear:* la app forma una consulta de búsqueda a la API de localización cada vez que el usuario escribe.
- *Leer:* se obtienen resultados (una lista) y se muestran.
- *Actualizar:* con cada carácter ingresado, se actualiza la lista de resultados.
- *Eliminar:* al cerrar la búsqueda o cuando el usuario borra texto, la lista anterior se descarta.
Servicios/APIs: Sí, uso de *MKLocalSearch* (Apple) para obtener sugerencias de direcciones/lugares. Alternativamente, podría usarse la API de Google Places. En ambos casos se trata de llamadas a servicios en la nube. Con MKLocalSearch, no se requiere clave de API ni manejo de autenticación explícita.
Autenticación: No requerida para Apple's local search. Para Google Places, se requeriría la API Key si se usara.
Formato: Los resultados vienen como objetos nativos (MKMapItem) con propiedades como name, coordinate, etc. Si fuera JSON (Google), vendrían en formato JSON con campos "predictions". Para la app, estos se convierten a strings para mostrar (nombre + dirección).
Costos: MKLocalSearch es gratuito; Google Places tiene costo después de ciertas cantidades.
Almacenamiento local: Por defecto, no se almacenan las sugerencias. Podríamos guardar *historial de búsquedas* en local (para mostrar "recientes"), pero en MVP se omitió. La no persistencia se debe a mantener la app ligera; además, la API provee sugerencias rápidamente, así que un historial no es imprescindible.

Pantalla de Detalle de Ruta:
Tipo de datos: Conjunto de instrucciones secuenciales (texto + quizás iconos) que describen la ruta. Operaciones:
- *Crear:* se crea esta lista tras obtener la ruta (MapKit internamente genera las instrucciones).
- *Leer:* la app lee las instrucciones de la ruta calculada y las presenta en una tabla/lista.
- *Actualizar:* si el usuario recalcula la ruta (por un cambio de destino o posición), la lista

se actualiza con las nuevas indicaciones. Durante la navegación, si implementáramos actualizaciones en tiempo real (fuera de MVP), podría marcar la instrucción actual completada y resaltar la siguiente.
- *Borrar:* cuando el usuario cierra la ruta o inicia una nueva búsqueda, las indicaciones previas se descartan de memoria.
Servicios/APIs: No hay llamadas adicionales aquí; la información viene ya incluida al consultar la ruta en MapKit. Es decir, *MKDirections* también proporciona las *steps* (pasos de la ruta). No necesitamos hacer otra conexión de red.
Autenticación: N/A (heredada de la llamada inicial de ruta).
Formato: Estructura de datos interna (por ejemplo, un arreglo de `MKRouteStep` en Swift). Cada paso tiene propiedades como *instructions* (String, ej. “Gira a la derecha en …”), *distance* (Double en metros), *transportType* (tipo de transporte, aquí .walking). La app formatea estos datos para mostrarlos.
Costos: Ninguno extra, ya cubierto en la llamada de ruta.
Almacenamiento local: No se guarda de forma persistente esta lista, dado que es temporal. Podría haber un caso de uso de guardar rutas favoritas por el usuario, pero MVP no lo contempla. Todas las rutas se generan on-demand.

Pantalla Módulo SOS:
Tipo de datos: Datos sensibles y en tiempo real. Involucra la ubicación actual del usuario (lat, lon) y el mensaje de SOS (texto predefinido). También utiliza el número de teléfono del contacto de emergencia. Operaciones:
- *Leer:* obtiene la ubicación actual vía GPS. También lee de almacenamiento local el número de contacto configurado.
- *Crear:* construye el contenido de la alerta (por ejemplo, compone un texto tipo “Ayuda, estoy en [enlace maps].” y lo prepara para envío). Si optamos por abrir la app de mensajería nativa, la creación en sí es delegar al sistema (no estamos nosotros enviando directamente, sino invocando un *MFMessageCompose* en iOS).
- *Actualizar:* esta pantalla en sí no tiene actualización continua, salvo quizás mostrar dinámicamente la dirección actual derivada de las coordenadas (geocodificación inversa) para referencia. Pero en MVP, con mostrar coordenadas o un link es suficiente.
- *Delete:* no aplica, salvo cancelar el SOS (lo cual simplemente no envía nada).
Conexión con servicios/APIs: Principalmente, usa servicios del dispositivo: GPS para ubicación. Puede usar *Reverse geocoding* (por ejemplo, `CLGeocoder`) para traducir coord a dirección legible – esto sería una llamada a Apple’s geocoding service, no indispensable pero útil. Para el envío de mensaje: la app puede usar la API nativa de SMS/teléfono. Esto no es una API web sino un servicio del sistema operativo. En iOS, para realizar un SMS desde la app se utiliza `MFMessageComposeViewController` (que requiere que el dispositivo tenga habilitado iMessage/SMS). Alternativamente, hacer una llamada telefónica se logra con el esquema `tel://` y delega al marcador telefónico. Ambos casos son integraciones con servicios del sistema, no de terceros en la nube.
Autenticación: No requiere login, pero el acceso a Servicios del sistema (ubicación, SMS) requiere permisos del usuario. Para ubicación, iOS pedirá permiso la primera vez. Para enviar SMS, iOS no pide un permiso por adelantado, pero sí requiere que el usuario confirme el envío dentro de la app de mensajes (por políticas de Apple, las

apps no pueden mandar SMS directamente sin interacción del usuario a menos que sea vía un servidor propio). En este MVP, esperamos que el usuario confirme el mensaje en el interfaz de mensajería, cumpliendo las reglas de la plataforma.
Formato: – *Ubicación:* coordenadas (que podemos convertir a un Google Maps link, por ejemplo `https://maps.google.com/?q=lat,lon`). – *Mensaje SOS:* texto plano. – *Número de teléfono:* string numérico. Si se usa la API de mensajería de Apple, se pasa estos datos al controlador de mensaje. Si se hiciera vía un servicio web (por ej. Twilio para SMS automático), entonces se enviaría una solicitud HTTP con un payload JSON/XML al servicio Twilio con el número y mensaje, pero eso incurriría en costos y complejidad (no usado en MVP).
Costos: En el planteamiento actual, no hay costo para la aplicación por enviar el SOS, ya que se utiliza la funcionalidad de SMS del propio usuario (el mensaje se envía con su plan móvil, igual que si lo escribiera él mismo). No usamos un servidor SMS de pago. Esto es importante porque delega el gasto al usuario (un SMS estándar) y evita manejar cobros en la app. Si optáramos por integrar un servicio de envío de SMS automatizado, habría un costo por mensaje (Twilio cobra centavos de dólar por SMS, etc.) además de requerir internet; por eso se prefirió la vía nativa.
Almacenamiento local: La pantalla SOS por sí misma no escribe nada permanente, pero depende de datos locales: el número de contacto de emergencia que el usuario configuró. Ese número se almacena en el dispositivo (en la pantalla de Contacto de Emergencia mencionada), probablemente en *User Defaults* por simplicidad, dado que es información pequeña y sensible (se podría cifrar, pero al ser un número de teléfono del propio usuario, no es altamente crítico; aún así iOS User Defaults es razonablemente seguro para este caso). La app usa ese dato. También podríamos almacenar un registro de "último SOS enviado a tal hora" para evitar spam de SOS (por ejemplo, no permitir enviar múltiples en pocos segundos) o para mostrar en la pantalla de confirmación. Esto sería un valor temporal en memoria o guardado si se quisiera persistir (no esencial).

Pantalla de Confirmación de SOS:
Tipo de datos: Sólo mensajes de UI (p.ej. "SOS enviado exitosamente"). Operaciones: *Leer* – muestra el mensaje. *Crear* – podría generar un registro local (log) con timestamp del envío. *Actualizar/Borrar:* No aplica. Servicios: No contacta nada externo; en este punto el SMS/llamada ya fue disparado en la acción anterior. Autenticación: N/A. Formato: Texto UI. Costo: Nulo. Local: Como se mencionó, opcionalmente anotar en local la hora de envío (si quisiéramos un histórico o para métricas internas). En MVP, no se lleva un historial formal, pero es una consideración para futuras versiones (por ejemplo, para mostrar al usuario "su último SOS fue hace X días").

Pantalla de Contacto de Emergencia:
Tipo de datos: Datos de usuario – nombre y teléfono. Operaciones:
- *Crear:* cuando el usuario guarda un contacto por primera vez, se crea la entrada (en almacenamiento local).
- *Leer:* cada vez que se abre la pantalla, se lee el contacto guardado (si existe) para mostrarlo.

- *Actualizar:* si el usuario edita y guarda de nuevo, se sobreescribe el dato existente.
- *Borrar:* podríamos permitir eliminar el contacto (dejándolo vacío), lo que en la práctica borra esos valores de almacenamiento o los marca como null.

Servicios/APIs: No usa servicios externos. Todo ocurre en la app. Podemos aprovechar frameworks iOS: por ejemplo, para ayudar al usuario a elegir un contacto podría integrarse el *Contact Picker* (acceso a la agenda del teléfono) – de esa forma no tendría que escribir el número manualmente. Si implementado, eso requeriría permiso de Contacts. En MVP podemos suponer que el usuario ingresa manual. Sin integraciones externas.

Autenticación: No. (Si se accede a la agenda, iOS pediría permiso de contactos, pero asumir ingreso manual en MVP evita eso).

Formato: Clave-valor simple. Podríamos usar NSUserDefaults (UserDefaults estándar) guardando, por ejemplo, `sosContactName` = "Juan" y `sosContactNumber` = "5512345678". Alternativamente, almacenar en un pequeño SQLite/CoreData, pero es exagerado para dos campos.

Costos: Ninguno, es funcionalidad local.

Almacenamiento local: Sí, este es un caso claro de uso de almacenamiento local. Guardamos el contacto en el dispositivo por persistencia – para que al cerrar y abrir la app se conserve. La razón es obvia: comodidad y necesidad (el usuario no querría ingresar su contacto de emergencia cada vez). Se decidió no almacenar este dato en ninguna nube o servidor (no hay backend propio), tanto por simplicidad como por privacidad; además, dado que no hay autenticación de usuario, no hay un perfil en servidor donde asociar ese dato.

Pantalla de Información/Acerca de:

Tipo de datos: Estáticos (textos informativos, enlaces). Operaciones: *Leer* – muestra el contenido. No hay creación ni edición por el usuario. Servicios: Podría contener un enlace a una página web (ej. política de privacidad en un sitio externo) que al tocarlo use Safari, pero la pantalla en sí no llama a APIs en segundo plano. Autenticación: No. Formato: texto HTML básico si se renderiza formateado, o simplemente UILabels con formato. Costos: No. Local: Los textos residen en la app, no hay almacenamiento dinámico aquí.

Dispositivos Soportados y Uso de Sensores

Dispositivos y tamaños de pantalla: *Conecta Fácil* está diseñada para funcionar en dispositivos iOS, principalmente iPhone. Se ha contemplado la variedad de tamaños de pantalla que existen en el ecosistema, desde iPhone SE (pantalla pequeña de 4.7") hasta los modelos Pro Max (~6.7") e incluso compatibilidad básica con iPad (pantallas mayores), aunque el enfoque principal es teléfonos. La interfaz utiliza un diseño adaptable (responsive) que se ajusta a diferentes resoluciones y orientaciones, garantizando una experiencia consistente en diversos dispositivos. Para lograr esto, se emplearon *Auto Layout constraints* en Xcode que permiten que los elementos (botones, mapas, listas) redimensionen o reubiquen según el espacio disponible. Por ejemplo, en pantallas pequeñas la barra de búsqueda y el botón SOS ocupan posiciones

optimizadas para no saturar la vista, mientras que en pantallas grandes pueden mostrarse con mayor separación.

En cuanto a la orientación, la app soporta tanto portrait (vertical) como landscape (horizontal), pero la orientación recomendada es portrait ya que la mayoría de las interacciones (búsqueda de rutas, lectura de instrucciones) se realizan más cómodamente de esa forma. No obstante, para la pantalla de mapa se consideró útil permitir landscape: en horizontal se aprovecha mejor el ancho para ver más tramo de ruta, y la interfaz se adapta (por ejemplo, la barra de búsqueda podría contraerse o repositionarse). Se garantiza que en landscape los elementos críticos (botón SOS, etc.) sigan accesibles y la vista no se vea extraña. El diseño adaptable implica considerar ambos modos en pruebas. Para otras pantallas como onboarding o perfil, se mantienen en portrait por simplicidad (incluso se podría fijar solo portrait en algunas secciones, pero idealmente se soporta todo para consistencia del sistema). En resumen, el diseño responsivo incluye tamaños flexibles, fuentes adaptativas y uso de controladores de interfaz estándar para facilitar la compatibilidad con múltiples dimensiones de pantalla.

Es importante mencionar que los dispositivos móviles difieren en capacidades de hardware desde tamaños de pantalla hasta sensores disponibles. *Conecta Fácil* hace uso de los siguientes sensores/componentes del dispositivo:

- GPS (Localización): Utiliza el receptor GNSS del iPhone a través del framework *Core Location* para obtener la ubicación geográfica del usuario en tiempo real. Esta localización es fundamental para ambas funciones principales de la app: calcular rutas (tomando la posición actual como origen) y proveer la posición en caso de emergencia. La app solicita permiso de localización *“mientras se usa”* al iniciarse por primera vez la funcionalidad de mapa. Solo tras la autorización del usuario, comienza a leer coordenadas. La conexión con el sensor GPS es continua (con nivel de precisión “Best” cuando se muestra la ruta, para mayor exactitud). Se maneja la lógica para no consumir energía excesiva: por ejemplo, detener actualizaciones cuando la app está en segundo plano (en MVP no se ejecuta en background prolongado). *¿Para qué?* Principalmente para anclar al usuario en el mapa automáticamente y para tener el punto de partida correcto en las rutas. En el módulo SOS, se toma la última ubicación conocida para enviarla. Cabe destacar que el uso de Localización implica considerar la privacidad: la app explica en la solicitud de permiso que es para *“mostrar rutas desde tu ubicación y compartirla en emergencias”*. Esto evita alarmar al usuario y cumple con lineamientos de la App Store (pedir solo lo necesario, con justificación).

- Micrófono (Voz): Aunque el MVP no explota ampliamente entrada o salida de voz, se planificó soporte para comandos de voz básicos en el futuro, principalmente orientados a accesibilidad. Por ejemplo, integrar el reconocimiento de voz de iOS para que un usuario pueda decir *“Hey Siri, abre Conecta Fácil y envía SOS”* o internamente implementar que con un comando de voz personalizado se active el SOS (pensando en casos donde el usuario no pueda manipular el teléfono físicamente). En la versión actual, no hay un

comando de voz propio dentro de la app, pero sí se puede aprovechar Siri Shortcuts: es posible registrar un acceso rápido de Siri para el SOS. De esta forma, el usuario configuraría una frase (“Conecta, ayuda”) que Siri reconocería y lanzaría la función SOS de la app. Esto usaría el sensor de micrófono del dispositivo en conjunto con el sistema Siri. La integración de Siri requiere adicionar en la app Intents de SiriKit (lo cual está en evaluación). Adicionalmente, la app es compatible con la funcionalidad de VoiceOver: aunque VoiceOver no es exactamente un sensor, es parte de accesibilidad – el usuario que navega con lectores de pantalla puede oír los nombres de botones (hemos etiquetado adecuadamente elementos UI para ello). Para fines de sensores, podemos mencionar que si en alguna versión se habilita comando por voz interno (usando *Speech Framework* de Apple para reconocimiento continuo), habría que pedir permiso de micrófono. En el MVP no se pide aún permiso de micrófono porque no se usa activamente dentro de la app (apoyarse en Siri no requiere que la app pida permiso, Siri misma lo maneja a nivel sistema). ¿Para qué se conectaría? – Principalmente para mejorar accesibilidad: imagínese un usuario caído que no puede alcanzar a tocar la pantalla, podría decir una frase de auxilio que la app entienda y dispare SOS. Esta es una funcionalidad propuesta a futuro por su claro valor inclusivo, aunque técnicamente compleja (requiere asegurarse de evitar activaciones falsas por ruido ambiental, etc.).

- Otros sensores: No se emplean otros sensores como tal en esta aplicación. Por ejemplo, no se utiliza cámara, ni acelerómetro/giroscopio directamente (aunque el mapa de Apple internamente puede usar el giroscopio para orientarse en modo de realidad aumentada, pero ese es un detalle interno de MapKit cuando uno activa la brújula, no algo que nuestro código maneje explícitamente). Tampoco se usan sensores de proximidad ni el lector de huella/FaceID (no hay login). El *compás/brújula* proveniente de Core Location podría usarse para orientar el mapa hacia donde mira el usuario, pero en MVP se dejó con orientación estándar (norte arriba) para simplicidad.

En síntesis, los dispositivos soportados son iPhones con iOS reciente, adaptando la UI a diferentes tamaños y orientaciones mediante diseño responsivo. Se aprovechan las capacidades hardware relevantes (GPS principalmente) para ofrecer la funcionalidad central, y se sientan bases para eventualmente integrar voz como método de interacción alternativo. Todo esto apunta a una experiencia móvil ubicua: el usuario puede llevar *Conecta Fácil* en cualquier iPhone y obtener acceso a la información de rutas y ayuda en cualquier lugar y momento, que es la promesa del cómputo móvil. El desarrollo consideró las variaciones de hardware para asegurar que la app corra de manera correcta y accesible donde sea requerida.

Wireframes

En esta sección se describen de 3 a 5 pantallas principales con su estilo final, incluyendo colores, tipografías, iconografía y textos de ejemplo. Las maquetas finales incorporan principios de diseño accesible e identidad gráfica consistente.

Figura 1: Pantalla principal con las opciones disponibles para el usuario, siendo la principal la que se muestra en primer lugar. Además se muestran las opciones de accesibilidad en esta pantalla.

La paleta de colores elegida combina tonos azul marino, turquesa y blanco, proyectando una imagen amigable y tecnológica a la vez. El azul turquesa se utiliza para resaltar elementos interactivos (por ejemplo, el botón de búsqueda, indicadores en el mapa), mientras que el rojo se reserva exclusivamente para la función SOS, dada su asociación universal con emergencias. El color de la letra de la mayoría de pantallas es blanco o muy claro para maximizar la legibilidad. Este esquema de alto contraste cumple con recomendaciones WCAG de accesibilidad, garantizando que el texto se distinga claramente del fondo, beneficiando a usuarios con baja visión.

La tipografía empleada es la fuente sistema de iOS (*San Francisco*), asegurando consistencia y optimización en diferentes tamaños de pantalla. Se usaron tamaños de fuente relativamente grandes para títulos y botones (>=17 pt para contenido principal), cumpliendo con las pautas de accesibilidad de iOS y facilitando la lectura para todo tipo de usuarios. Además, se habilitó la propiedad *Dynamic Type* en los componentes de texto pertinentes, de modo que si un usuario configura letras más grandes en su iPhone (Accesibilidad > Texto más grande), la app ajuste sus labels en consecuencia, manteniendo el diseño fluido.

Iconografía: Para los íconos se optó por los SF Symbols de Apple, que son iconos vectoriales nativos con estilo consistente con iOS. Estos íconos tienen la ventaja de ajustarse automáticamente de grosor según la fuente del sistema y soportar las mismas propiedades dinámicas (asegurando claridad en distintos tamaños). Además, los SF Symbols incluyen etiquetas accesibles ya integradas, lo que facilita la compatibilidad con VoiceOver (lector de pantalla anunciará “botón de búsqueda” correctamente, por ejemplo).

Figura 2: Pantalla de horarios del transporte público con dirección a las sedes del mundial, se muestran los detalles de cada alternativa así como los costos.

En este flujo de pantallas se muestra la información del transporte público cuyas rutas permiten llegar a las sedes del mundial, con fondo color azul y letras blancas para su mayor legibilidad, se muestran las líneas, horarios de servicio, frecuencia estimada de cada tren o camión, así como las estaciones o paradas destino. Al final del listado de alternativas se muestra información sobre la accesibilidad de las líneas, el costo del servicio de transporte público y notas importantes.

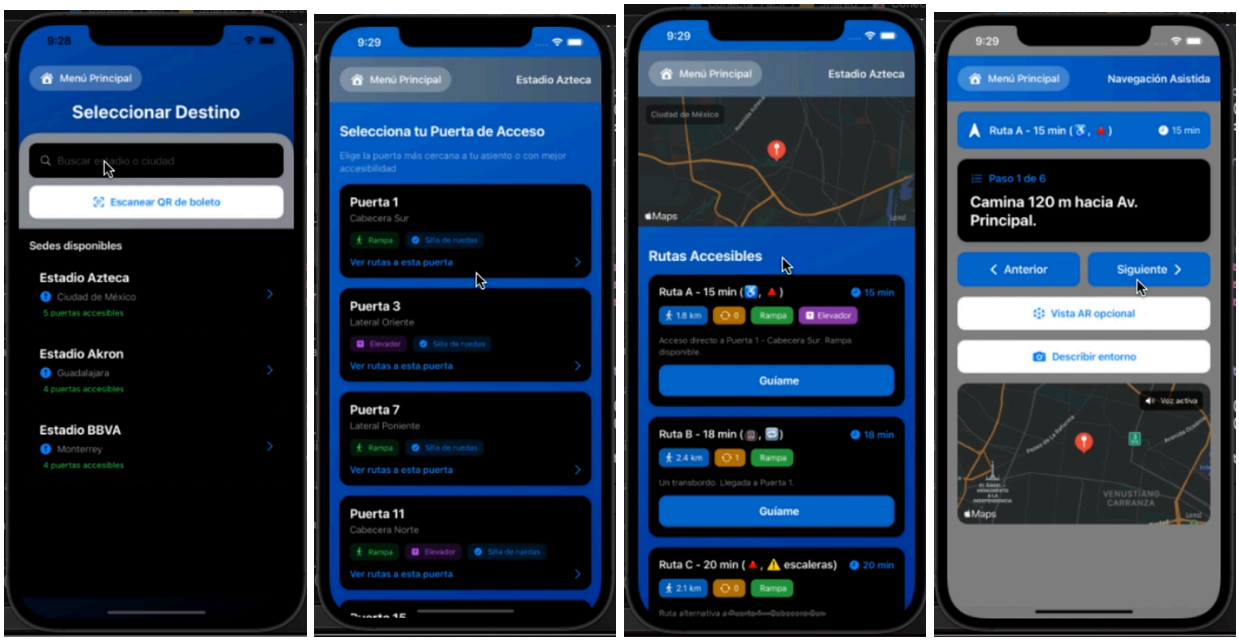

Figura 3: Pantalla de instrucciones de ruta. Se muestra la lista de pasos con texto claro en fondo negro e iconos de flecha azules.

En la pantalla de Detalle de Ruta, el diseño final consiste en un listado con cada paso mostrando un icono (flecha recto, giro a la izquierda, etc.), la distancia y la instrucción. Cada item está contenido en una tarjeta o celda de fondo ligeramente gris muy claro para alternar visibilidad. El paso actual (si se estuviera navegando) se resalta en un tono azul más intenso de la paleta. Las fuentes en esta vista son de 16 pt para instrucciones y 14 pt para distancias, proporcionando jerarquía visual. Esta pantalla incluye una barra de navegación superior con el título “Indicaciones” y un botón para regresar al mapa (arrow back), consistente con la navegación jerárquica de iOS.

Figura 4: Pantalla de funciones de emergencia, con el botón SOS destacado y en mayor tamaño en comparación a los demás botones de contactos de emergencia, hospitales cercanos y protocolos rápidos.

La pantalla SOS presenta un fondo quizás ligeramente atenuado (blur sobre el mapa actual, para mantener contexto de dónde está, o un patrón de emergencias). En el centro, un gran botón redondo rojo con el texto “SOS” en blanco y un icono de sirena. Este botón es deliberadamente grande (unos 120pt de diámetro) para ser fácilmente pulsable en situación de emergencia, cumpliendo también los lineamientos de objetivos táctiles mínimos (Apple recomienda al menos 44x44pt, nuestro botón excede eso holgadamente). Al presionar el botón SOS, se realiza la solicitud a los servicios de emergencia y de ser posible, se muesta la información de la ambulancia y el hospital que atienden la solicitud, así como la ubicación en el mapa del origen de la emergencia

y en caso posible, la ubicación de la ambulancia. Toda la interfaz de la pantalla SOS está pensada con *el menor ruido posible*: pocos elementos, lenguaje claro (“Enviar alerta ahora a [Contacto]”), para que en una situación de estrés el usuario no tenga que pensar demasiado. El diseño final también contempla retroalimentación háptica: al iniciar el envío SOS, el teléfono vibra brevemente (feedback físico) para confirmar la acción.

En términos de accesibilidad en el diseño visual, se validó la combinación de colores con herramientas de contraste para asegurarse que incluso personas con daltonismo pudieran distinguir el botón SOS (rojo puro con texto blanco) del resto de la interfaz.

Figura 5: Pantalla de modo turista, se muestran recomendaciones de horarios, consejos sobre los boletos y entradas, información sobre el transporte y comida así como recomendaciones de seguridad y tips para asistentes extranjeros.

Esta pantalla presenta un fondo de color turquesa con degradado en negro, letras blancas e íconos que ayuden a comprender la información mostrada asociada a un color, para desplegar la información es necesario hacer tap sobre la categoría, lo mismo para ocultarla, hay un botón en la esquina superior derecha para regresar al menú principal.

Finalmente, la estética general equilibra simplicidad y funcionalidad. No se sobrecargó la interfaz con adornos innecesarios; cada elemento tiene una razón de ser. Esto sigue el principio de *“diseño universal”*: un UI limpio beneficia a todos los usuarios, especialmente a aquellos con dificultades cognitivas o visuales, quienes se orientan mejor con una interfaz clara y coherente. Las maquetas finales fueron hechas en Figma (por ejemplo) y adjuntadas también como imágenes, sirviendo de guía exacta para la implementación en código.

Lenguaje de Programación y Herramientas de Desarrollo

Para el desarrollo de *Conecta Fácil* se empleó Swift como lenguaje de programación y el entorno de desarrollo Xcode, orientado a la plataforma iOS. Esta elección se justifica tanto por requisitos del proyecto (el "cliente"/profesor solicitó específicamente una aplicación iOS nativa) como por beneficios técnicos y estratégicos:

- Requerimiento de plataforma: El proyecto desde su inicio estuvo enfocado a iOS, por lo que desarrollar nativamente en el ecosistema Apple era la ruta natural. Swift es el principal lenguaje soportado por Apple para iPhone/iPad, habiendo desplazado a Objective-C en los últimos años. Cumplimos así con las expectativas de utilizar herramientas estándares de la plataforma.

- Ventajas de Swift: Swift es un lenguaje moderno, seguro y eficiente, diseñado por Apple para integrarse con frameworks Cocoa/Cocoa Touch. Uno de sus puntos fuertes es la prevención de errores comunes de programación: su sintaxis y sistema de tipos evitan muchos *crashes* (por ejemplo, manejo seguro de variables opcionales en vez de punteros nulos). Esto resulta en código más confiable y fácil de depurar, esencial en una app donde la estabilidad es crítica (imaginemos un fallo en medio de una emergencia; debemos evitarlo). Swift también favorece un desarrollo ágil: su sintaxis concisa y expresiva permite escribir funcionalidad con menos líneas de código en comparación con Objective-C. Durante el proyecto, comprobamos que funcionalidades como las notificaciones de cambios de ubicación o el dibujo de rutas en el mapa se implementaron rápidamente gracias a bibliotecas nativas de Swift (por ejemplo, `MapKit` con Swift tiene APIs muy directas). Adicionalmente, Swift ofrece alto rendimiento: su código compilado se aproxima en velocidad al de C/Obj-C pero con optimizaciones modernas de compilador. Esto nos da la tranquilidad de que la app podrá ejecutar cálculos ligeros (formatear instrucciones, por ejemplo) sin penalización perceptible para el usuario.

- Uso de Xcode e integraciones nativas: Xcode, la IDE oficial de Apple, proporciona un conjunto de herramientas robustas: Interface Builder para diseñar storyboards y interfaces visualmente (lo que aceleró el prototipado de pantallas), simuladores de dispositivos para pruebas en distintas versiones de iPhone, y herramientas de depuración avanzadas (inspección de memoria, CPU, etc.). La adopción de Xcode permitió también utilizar SwiftUI Previews para componentes de interfaz rápidos (aunque principalmente se usó UIKit para tener mayor control en este proyecto, algunas vistas simples pudieron previewearse). Al mantenernos en el stack nativo, pudimos aprovechar componentes estándar de iOS – por ejemplo, los controladores de alerta, las tablas nativas, etc., que ya traen comportamientos accesibles por defecto. Esto reduce la carga de programar detalles desde cero y asegura que la navegación y apariencia se sientan "nativas", cumpliendo con expectativas de usuarios de iPhone. Además, usando herramientas nativas es más sencillo cumplir lineamientos de la App Store.

- Comparación con enfoques alternativos: Se consideró brevemente la posibilidad de usar un framework multiplataforma (como React Native o Flutter) por curiosidad, pero se descartó por varias razones: (1) Requerimiento explícito de iOS nativo – el cliente quiere posiblemente evaluar habilidades en Swift/Xcode; (2) Las integraciones de hardware (GPS, etc.) suelen ser más directas en nativo, evitando puentes; (3) El rendimiento y look & feel de un app nativa tiende a ser superior, especialmente en detalles de accesibilidad y adaptabilidad. De hecho, en el curso se revisó que los lenguajes nativos (Swift/Objective-C en iOS, Java/Kotlin en Android) ofrecen ventajas en aprovechamiento del hardware y consistencia de interfaz, a diferencia de soluciones híbridas basadas en webviews.

- Accesibilidad y frameworks: Otra razón para elegir Swift/UIKit fue la disponibilidad de frameworks como *Accessibility API* de iOS que se integran fácilmente. Por ejemplo, marcar elementos con `accessibilityLabel` o usar componentes nativos garantiza que tecnologías como VoiceOver puedan interactuar con la app sin configuraciones extra. Esto se alinea con el objetivo de crear una app inclusiva. En un entorno híbrido, probablemente se hubieran tenido que manejar más casos manualmente.

- Distribución en App Store: El uso de Swift/Xcode facilita la preparación para la App Store, ya que el proyecto resultante (archivo .ipa) está listo para subir mediante App Store Connect sin configuraciones especiales. La integración con herramientas de la App Store (como TestFlight para pruebas beta) es directa con Xcode. Esto nos posiciona bien para el paso de publicación (ver siguiente sección).

Requisitos para la Publicación en App Store

Publicar *Conecta Fácil* en la App Store de Apple requiere cumplir una serie de requisitos técnicos y normativos. A continuación se enumeran las consideraciones principales que preparamos para la eventual distribución pública de la aplicación:

- Cuenta de Desarrollador Apple: Es necesario tener una cuenta de desarrollador activa en Apple Developer Program. Ya se ha tramitado una cuenta a nombre del equipo (o de la institución académica) con el rol de *Team Agent*. Esta cuenta implica un costo anual de $99 USD (aproximadamente $1,700 MXN), lo cual se debe contemplar en el presupuesto si la app fuera a ser publicada. Con esta cuenta, se obtiene acceso a App Store Connect para gestionar la app y a certificados necesarios para firmar el binario.

- Identificador único y firma: Antes de publicar, la app debe tener un *Bundle Identifier* único (por ejemplo, mx.university.conectafacil) y estar firmada con un certificado de distribución válido. Configuramos el proyecto en Xcode con estos datos y generamos un archivo .IPA firmado listo para envío. Se verificó que no haya errores de firma ni uso de *entitlements* no permitidos (como iCloud, push,

etc., que no usamos). Errores en firma o perfiles de aprovisionamiento son causas comunes de rechazo, por lo que hicimos pruebas en TestFlight para validar.

- Directrices de la App Store: Apple tiene guías estrictas (App Store Review Guidelines) que la app debe cumplir. Hemos revisado los puntos relevantes, asegurando por ejemplo que:

- No se usa contenido prohibido (violencia extrema, lenguaje inapropiado, etc.) – nuestra app es utilitaria y apta para todo público.
- No hay uso de API privadas del sistema – solo usamos APIs públicas aprobadas.
- La descripción de la app en la tienda será honesta y clara, evitando *keywords stuffing* o alegaciones engañosas (cumpliendo la guía de contenido).
- Al implementar la función SOS, confirmamos que no viola reglas: Apple no prohíbe que se envíen SMS a un contacto elegido por el usuario, pues requiere interacción del mismo (lo cual es permitido). En cambio, hay lineamientos sobre llamadas automáticas a emergencias (911) – eso está prohibido. *Conecta Fácil* no llama directamente a 911, solo a contactos de confianza; esto es importante para no ser rechazada.
- También, se revisó que la app no quede en un *loop* de requerir permisos sin explicar (Apple puede rechazar apps que piden muchos permisos innecesariamente). En nuestro caso, pedimos solo Localización, y justificadamente. Igualmente, seguimos la recomendación de Apple de incluir el texto de propósito en el `Info.plist` para el permiso de Localización: “Conecta Fácil necesita acceder a tu ubicación para calcular rutas accesibles y permitirte enviar tu ubicación en caso de emergencia”. Con esto, evitamos alertas de “Permisos no justificados” durante la revisión.

- Política de Privacidad: Apple exige que toda app que recopila datos del usuario o tenga alguna funcionalidad de registro tenga una política de privacidad disponible. Si bien *Conecta Fácil* no realiza registro de cuentas ni envía datos a servidores propios, sí utiliza información personal (ubicación, contacto de emergencia). Por ello, hemos preparado un breve documento de Política de Privacidad donde se explica qué datos se usan y cómo (por ejemplo: *“La ubicación del usuario se utiliza localmente para trazar rutas y en mensajes SOS bajo comando del usuario; no almacenamos ni compartimos esta ubicación en ningún servidor.”*). Esta política estará accesible mediante una URL pública. De hecho, App Store Connect solicita proporcionar la URL de la política de privacidad incluso para apps sin login. Tenemos planeado alojarla en un sitio gratuito o en la página de la universidad. Asegurarse de esto previene un rechazo inmediato, ya que Apple rechaza apps sin política cuando corresponde.

- Ficha de la App en App Store Connect: Se deberá completar la metadata de la app: nombre (ej. “Conecta Fácil”), subtítulo (una frase breve), descripción

(resaltando las funciones clave en lenguaje de marketing ligero), categoría (muy probablemente “Utilidades” o “Navegación”), y especificar cosas como: edad recomendada (likely 4+ ya que no hay contenido sensible), idioma (español), región (inicialmente México), etc. También contact info del desarrollador (un correo de soporte). Todo esto lo tenemos preparado. Cuidaremos especialmente las capturas de pantalla: Apple requiere subir screenshots de la app en varias resoluciones de dispositivos (ej. 6.7” para iPhone Pro Max, 5.5” para formatos antiguos). Hemos tomado capturas de las maquetas finales usando el simulador de Xcode para cumplir con estas resoluciones, asegurándonos de que la interfaz se vea completa (sin contenido cortado). Las imágenes promocionales son de alta calidad y muestran la funcionalidad (Apple puede rechazar capturas que sean meros pantallas de carga o no muestren la app real). También preparamos un ícono de la app en las dimensiones requeridas (1024x1024 px para el App Store, más tamaños menores para los assets internos de la app). El ícono de *Conecta Fácil* sigue la estética del proyecto: un círculo azul turquesa con un símbolo de persona y GPS estilizado, simple y reconocible incluso en tamaño pequeño.

- Permisos declarativos en App Store: Desde iOS 14, Apple solicita en App Store Connect un formulario de *Privacy Nutrition* donde se declara qué datos recoge la app y con qué propósito. En nuestro caso, hay que declarar que se obtiene *“Precise Location”* del usuario, con propósito de funcionalidad principal (navegación) y de seguridad (SOS). Dado que no compartimos esa data con terceros, marcamos que los datos no salen del dispositivo. También posiblemente declaremos que el usuario voluntariamente introduce datos de Contactos (el número de emergencia) pero aclarando que se queda local. Esto es parte del proceso de publicación y es obligatorio para que Apple apruebe la app, pues muestran esa “ficha de privacidad” a los usuarios en la App Store.

- Pruebas finales y proceso de revisión: Antes de enviar a revisión, probamos la app en un dispositivo físico (iPhone real) para verificar que todos los permisos y funciones se comporten correctamente en entorno real. Especialmente, se probó el flujo de Localización + Mapas + SMS, ya que involucra servicios del sistema. Tras asegurarlo, subiremos la app. Apple realizará una revisión manual que suele tardar 1-3 días. Durante esta, podrían verificar: navegación básica, que al pedir ubicación sale el prompt con la razón adecuada (lo harán en diferentes escenarios), podrían enviar un SOS de prueba (por eso hemos contemplado un modo demo o que envíe a un número de prueba si la cuenta de Apple lo sugiere, aunque normalmente revisan que se abra correctamente el panel de SMS y no más). Si los revisores encuentran algún problema (por ejemplo, la app crashea sin internet al buscar ruta), nos rechazarían con una notificación detallando el punto. Por ello, incorporamos en la app checks para ausencia de internet (mostrar alertas amigables) y mensajes de error si no hay GPS. Minimizar esos casos reduce probabilidades de rechazo.

- Adecuaciones específicas de publicación: Un detalle final: Apple requiere que la app soporte al menos los dispositivos iPhone X/11 con *notch* adecuadamente. Nos aseguramos de que ninguna vista quede oculta bajo las esquinas redondeadas o notch (usando *Safe Areas* en el diseño). También se añadió un pequeño texto en la pantalla de About indicando copyright y “Esta app fue desarrollada como proyecto académico” para transparencia. Aunque no es obligatorio, añade contexto y quizás evita confusiones (por ejemplo, si Apple se pregunta por qué hay 4 nombres de integrantes, queda claro que son desarrolladores).

Equipo de Trabajo y Roles

El desarrollo de *Conecta Fácil* fue llevado a cabo por el equipo Lambda, conformado por cuatro integrantes. Cada miembro asumió uno o más roles durante el proyecto, cubriendo las distintas áreas necesarias para completar exitosamente una aplicación móvil profesional. A continuación se describen los roles principales involucrados, alineados con estándares industriales:

- Project Manager / Product Manager: Responsable de la gestión integral del proyecto y la visión del producto. En nuestro equipo, este rol se encargó de planificar las tareas, establecer el cronograma de desarrollo y asegurarse de que los requisitos del “cliente” (profesor y usuarios finales) se cumplieran. El PM tradujo los objetivos del proyecto en un lenguaje entendible para el equipo técnico y de diseño, actuando como puente entre la visión del producto y la ejecución. También se ocupó de remover obstáculos, coordinar reuniones de seguimiento semanales y mantener al equipo enfocado en el MVP, evitando la *sobrecarga de alcance*. En un entorno profesional, el Project Manager vela por la calidad y entrega oportuna del producto, y en este proyecto ejercimos esa disciplina mediante herramientas ágiles (breves reuniones diarias, uso de Trello para asignar tareas, etc.).

- Diseñador UX/UI: Encargado de la experiencia de usuario (UX) y de la interfaz visual (UI) de la aplicación. Este rol tomó las ideas iniciales y las materializó en wireframes y posteriormente en prototipos de alta fidelidad. En la etapa de wireframing, definió la arquitectura de la app (jerarquía de pantallas, navegación) y cómo debía sentirse la interacción, asegurándose de que la aplicación fuera intuitiva y agradable de usar. Luego, trabajó en el diseño gráfico: elección de colores, tipografía, creación del logo e íconos personalizados cuando fue necesario. Siempre con un enfoque en accesibilidad, el diseñador UX/UI verificó que elementos como botones y textos siguieran las guías (por ejemplo, botones de buen tamaño, contrastes adecuados, etc.). Este rol también creó los assets exportables (imágenes @2x/@3x) para la implementación. Gracias a su trabajo, *Conecta Fácil* tiene una apariencia profesional y una usabilidad probada con usuarios de prueba antes del desarrollo.

- Desarrollador (Mobile Developer): Responsable de codificar la aplicación en Swift y construir la funcionalidad técnica. Dado el tamaño del equipo, los miembros compartieron tareas de desarrollo, abarcando tanto el *front-end* (la parte de la app que interactúa con el usuario: vistas, controladores, animaciones) como la potencial *lógica de back-end* (integración con servicios, manejo de datos). En un equipo grande podrían separarse en desarrollador iOS front-end (enfocado en implementar las pantallas tal como el diseñador las concibió, garantizando fluidez en transiciones y manejo correcto de gestos) y desarrollador back-end/servicios (encargado de la lógica de conexión con APIs, bases de datos, etc.). En nuestro caso, los desarrolladores tuvieron que ser *full-stack mobile*, ocupándose de todo el flujo interno: desde obtener la ubicación GPS hasta parsear una respuesta de ruta y dibujarla en pantalla. Este rol implicó escribir código limpio y eficiente, siguiendo patrones de diseño donde se pudo (por ejemplo MVC, delegados para comunicar eventos como la llegada de nuevos datos de mapa a la UI). Los desarrolladores también realizaron pruebas unitarias básicas de componentes clave (como la función que formatea el mensaje SOS), para asegurar la calidad. Cabe mencionar que uno de los desarrolladores asumió la integración continua: configurando el proyecto en un repositorio Git y automatizando builds de prueba en TestFlight.

- QA Tester (Quality Assurance): Encargado de asegurar la calidad del producto mediante pruebas sistemáticas. En fase final, un miembro del equipo tomó el rol de QA Tester, realizando pruebas funcionales exhaustivas en la app. Se elaboró un plan de pruebas cubriendo los casos de uso principales y escenarios de esquina: búsqueda de ruta válida, búsqueda de dirección inexistente (¿la app maneja bien un “no se encontraron resultados”?), envío de SOS con contacto configurado vs sin configurar (debe advertir al usuario que falta registrar contacto), comportamiento sin internet (mostrar alerta y no crashear), etc. El QA Tester documentó los bugs encontrados y trabajó con los desarrolladores para solucionarlos. También se probaron distintas configuraciones de iOS (por ejemplo, ¿qué ocurre si el usuario deniega permiso de ubicación? – la app debe manejarlo mostrando una pantalla informativa que lo necesita). Este rol es crucial porque detecta fallos antes de que los usuarios finales lo hagan. En un entorno profesional, QA además se aseguraría de pruebas de rendimiento y seguridad; dada la escala del proyecto, nos concentramos en pruebas funcionales manuales y algo de pruebas de usabilidad (pedimos a un par de personas externas que usaran la app y dieran feedback, lo cual sirvió para pequeños ajustes).

- Otros roles considerados: Si bien los cuatro roles anteriores cubrieron las necesidades inmediatas, es útil mencionar otras figuras que existen en equipos móviles completos. Por ejemplo, un Business Analyst/Technical Writer podría haber formalizado requisitos y mantenido la documentación; en nuestro equipo esa tarea la asumió en parte el PM. Un DevOps Engineer no fue necesario dado que no tenemos infraestructura que desplegar continuamente (no hay servidor

backend), pero de haberlo, se encargaría de integrar la app con servicios en la nube y automatizar builds. Un rol de Marketing o Sales entra usualmente más adelante para promocionar la app; en contexto académico lo mencionamos solo teóricamente. Por último, un Scrum Master (si aplicamos metodología Scrum) habría facilitado la autogestión del equipo; esa función recayó igualmente sobre el PM de facto, aunque en un equipo de 4 la gestión fue ligera.

Estimación de Tiempo y Costos del Proyecto

Para llevar a cabo *Conecta Fácil* se realizó una estimación tanto del tiempo necesario en cada fase del ciclo de desarrollo, como de los costos asociados, en caso de proyectar el desarrollo de manera profesional. A continuación, se presenta un resumen en semanas para el cronograma, junto con un cálculo de costos estimados en pesos mexicanos (MXN):

Fase del Proyecto	Duración (semanas)	Costo estimado (MXN)
Levantamiento de Requerimientos y Diseño (UX/UI) *(Investigación inicial, bocetos, wireframes, maquetación)*	2 semanas	$15,000 MXN
Desarrollo de la Aplicación (Implementación MVP) *(Programación Swift, integración de mapas, funcionalidades SOS)*	8 semanas	$80,000 MXN
Pruebas e Iteración (QA & Refinamiento) *(Pruebas funcionales, corrección de bugs, optimizaciones finales)*	2 semanas	$15,000 MXN
Total Desarrollo MVP	~12 semanas	$110,000 MXN
Mantenimiento Post-lanzamiento *(Corrección de errores en producción, actualizaciones menores, soporte a*	*Continuo* (estimado 4-6 semanas de esfuerzo distribuido en el año)	~$20,000 MXN / año

Fase del Proyecto	Duración (semanas)	Costo estimado (MXN)
usuarios durante el primer año)		

Detalle de la estimación de tiempo: Se planificaron 12 semanas en total (aproximadamente 3 meses) para lograr el MVP funcional. En las primeras 2 semanas se concentraron las actividades de definición (reuniones con “stakeholders” para aclarar requerimientos, benchmark de apps similares, etc.) y diseño de experiencia. Gracias a contar con un diseñador dedicado, se generaron rápidamente los prototipos y se validaron con el equipo antes de codificar. Las siguientes ~8 semanas constituyeron el grueso del desarrollo en Xcode: aquí los desarrolladores dividieron tareas (uno enfocándose en la funcionalidad de rutas/mapa y otro en el módulo SOS y pantallas de ajustes, por ejemplo) y avanzaron iterativamente. Se adoptó un enfoque ágil con entregas parciales cada semana (al finalizar la semana 4 ya teníamos una primera versión integrando mapa y búsqueda básica; al finalizar semana 6, el SOS implementado en entorno de prueba; etc.). Las últimas 2 semanas se destinaron a QA y pulido: en la semana 9 se llevó a cabo una ronda de pruebas completa encontrando algunos bugs (por ejemplo, la app se cerraba al buscar una ubicación sin internet – caso que se arregló manejando la excepción), la semana 10 se corrigieron esos detalles. La semana 11 se repitieron pruebas y se ajustaron detalles de UI (alineaciones, textos), y en la semana 12 se dio por concluido el MVP, generando el build final y su documentación. Este cronograma se alineó con lo previsto; si acaso, el desarrollo tomó unos días más en integrar Siri Shortcuts (una funcionalidad menor que decidimos incluir al final), pero logramos encajarlo sin retrasar la entrega.

Detalle de costos estimados: La tabla asume un escenario profesional. Aunque en el contexto académico no hubo un costo real (el trabajo fue realizado por estudiantes), es útil calcular el valor económico aproximado. Se consideró: un diseñador UX/UI podría cobrar alrededor de $300 MXN por hora; en dos semanas (~80 horas) eso serían $24,000 MXN, pero suponiendo un esquema más ajustado (o dedicación parcial) se estimaron $15k para esa fase. Para desarrollo, se estimó un equipo equivalente a 1 desarrollador senior o 2 junior a tiempo completo durante 2 meses. En el mercado mexicano, la tarifa promedio de un desarrollador móvil puede rondar $180-$250 MXN/hora. Tomando un promedio de $200 MXN/hora, 2 desarrolladores, 8 semanas (~320 horas en total), resulta ~$64,000 MXN; agregando margen para pruebas e infraestructura, redondeamos a $80k. La fase de QA y refinamiento se calculó en $15k, asumiendo un tester por 2 semanas + horas de desarrollador para corregir issues. En total, el costo de desarrollo del MVP se proyecta en torno a $110,000 MXN, cifra consistente con estimaciones de mercado para una app básica en una plataforma (según un estudio de Seedup, una app móvil básica/MVP en México suele costar entre $50k y $150k MXN con ~6-10 semanas de desarrollo).

Para mantenimiento, se contempló aproximadamente un 15% del costo inicial por año. Esto cubriría: resolver bugs que aparezcan tras lanzamiento, actualizar la app a nuevas versiones de iOS (por ejemplo, si sale iOS 17 con cambios que afectan el

funcionamiento), y eventualmente incorporar pequeñas mejoras solicitadas por usuarios. Se calculó ~$20k MXN anuales, equivalentes quizá a unas 3-4 semanas de trabajo de un desarrollador distribuidas en el año para mantener la app estable y vigente. Cabe resaltar que esta cifra puede variar mucho dependiendo del éxito de la app y las solicitudes de nuevas funciones; pero como mantenemos el scope limitado en MVP, no anticipamos mantenimiento costoso en el primer año.

Consideraciones no monetarias: Además del costo directo, vale la pena mencionar los recursos y herramientas utilizados gratuitamente: el IDE Xcode no tuvo costo (es gratuito), las APIs de Apple utilizadas tampoco implicaron costo variable. El equipo hizo uso de servicios gratuitos para coordinación (GitHub repositorio privado, Google Drive para compartir documentos de diseño, etc.). Únicamente el programa de desarrollador de Apple implicaría un costo de $99 USD anual como ya se señaló, que se puede imputar al presupuesto de publicación más que al de desarrollo. Si el proyecto se implementara comercialmente, quizás se invertiría en más pruebas de dispositivos (comprar varios modelos de iPhone para probar, etc.), pero en nuestro caso utilizamos simuladores y uno o dos dispositivos reales disponibles.

Por último, en términos de relación tiempo-costo, el proyecto fue relativamente acotado: 3 meses para un MVP. Esto fue posible gracias a la definición clara de alcances (enfocarnos en rutas y SOS, dejando extras fuera) y a la reutilización de componentes existentes (APIs de mapas en lugar de implementar nuestro motor, librerías nativas para UI en lugar de código custom). En un proyecto comercial, esta eficiencia se traduciría en ahorro de costos y una llegada más rápida al mercado con un producto funcional para validar con usuarios.

Referencias

Adaptive. (2023). *Publicar una App en App Store y Google Play*. Recuperado de https://adaptivetech.es/actualidad-digital/publicar-app-app-store-y-google-play/

Apple. (n.d.). *Acerca de la privacidad y la función Localización en iOS, iPadOS y watchOS*. Soporte de Apple. Recuperado el 19 de noviembre de 2025, de https://support.apple.com/es-us/102515

Bautista, A. (2024, 21 de noviembre). *Aplicaciones móviles accesibles: Experiencias inclusivas desde el principio*. Tu Web Accesible. Recuperado de https://www.tuwebaccesible.es/aplicaciones-moviles-accesibles-e-inclusivas/

Canle Fernández, E. (2025, 21 de febrero). *Ventajas y desventajas del lenguaje Swift*. Tokio School. Recuperado de https://www.tokioschool.com/noticias/swift-ventajas-desventajas/

Pérez de Lara Domínguez, M. (2025a). *Cómputo Móvil – Overview* [Presentación]. Curso Cómputo Móvil, Facultad de Ingeniería, UNAM.

Pérez de Lara Domínguez, M. (2025b). *Tecnologías, ambientes de desarrollo y mercados* [Presentación]. Curso Cómputo Móvil, Facultad de Ingeniería, UNAM.

Platzi. (n.d.). *Roles y Estructura de un Equipo de Desarrollo Móvil Eficiente*. Platzi (Blog curso Ciclo de Vida de Apps). Recuperado de https://platzi.com/cursos/ciclovida-apps/roles-y-equipos-en-desarrollo-mobile/

Seedup. (s.f.). *¿Cuánto Cuesta Desarrollar una App en México?* (Infografía). Recuperado el 19 de noviembre de 2025, de https://seedup.mx/cuanto-cuesta-desarrollar-una-app-en-mexico

Steiner, J. (2025, 24 de octubre). *Guía de pruebas de accesibilidad para aplicaciones móviles*. Digital.ai. Recuperado de https://digital.ai/es/catalyst-blog/mobile-accessibility-testing

Wikipedia. (n.d.). *Producto viable mínimo*. Recuperado el 19 de noviembre de 2025, de https://es.wikipedia.org/wiki/Producto_viable_m%C3%ADnimo

Nota adicional: Los wireframes completos de *Conecta Fácil* se encuentran disponibles como anexos en la carpeta de entrega (archivo **wireframes.zip**).