Propósito de la aplicación

1. Introducción: Urbis Cádiz (Enfoque en Accesibilidad)

1.1. Propósito: Una Ciudad sin Barreras

Urbis Cádiz nace como una herramienta de transformación social orientada a mejorar la **movilidad urbana de personas con discapacidad física**. El proyecto se basa en la idea de "mapeo colaborativo" para identificar puntos críticos de la infraestructura urbana que impiden el libre movimiento de ciudadanos con sillas de ruedas, andadores o carritos.

1.2. Problemática y Solución

- **El Problema:** La falta de información actualizada sobre el estado de la accesibilidad en las calles. Una rampa en mal estado o una acera estrecha puede suponer el fin de un trayecto para una persona con discapacidad física.
- **La Solución:** Crear un **Inventario Vivo de Accesibilidad**. Al permitir que los usuarios reporten barreras con una foto y coordenadas exactas, la app genera un mapa de advertencias que ayuda tanto al colectivo de discapacitados como al Ayuntamiento para priorizar las reparaciones urgentes.

2. Tecnologías Utilizadas

Para este proyecto se ha seleccionado un stack tecnológico moderno que garantiza rendimiento y escalabilidad:

- **Kotlin:** Lenguaje de programación oficial para Android, aprovechando su seguridad frente a nulos y su sintaxis concisa.
- **Jetpack Compose:** Framework declarativo para la interfaz de usuario, permitiendo crear pantallas reactivas con menos código.
- **Firebase (BaaS):** * **Auth:** Gestión segura de identidades.
    - **Firestore:** Base de datos NoSQL para sincronización en tiempo real.
    - **Storage:** Almacenamiento de evidencias fotográficas.
- **Google Maps SDK:** Motor cartográfico para la visualización geoespacial de los datos.
- **Coil:** Librería para la carga asíncrona de imágenes mediante memoria caché.

3. Arquitectura del Sistema

La aplicación sigue el patrón **MVVM (Model-View-ViewModel)** combinado con los principios de **Clean Architecture**, dividiéndose en tres capas:

3.1. Capa de Presentación (UI)

Utiliza **Screens** (Compose) que observan el estado de los **ViewModels**. No hay lógica de negocio en las vistas; estas solo reaccionan a los cambios de estado y envían eventos de usuario.

3.2. Capa de Dominio (Interfaces)

Se han definido interfaces como IAuthRepo para desacoplar la lógica de la tecnología. Esto permite que la aplicación no dependa directamente de Firebase, facilitando futuros cambios de proveedor de datos.

3.3. Capa de Datos (Repositories)

Los **Repositorios** (AuthRepository, IncidenciaRepository) son la "única fuente de verdad". Gestionan la comunicación con las API de Firebase y transforman los datos crudos en objetos de dominio listos para ser usados por la UI.

Este es el núcleo de tu **Documentación Técnica**. Está diseñado para que cualquier evaluador entienda que no solo has programado, sino que has tomado decisiones de ingeniería de software sólidas.

1. Introducción y Justificación del Proyecto

1.1. Concepto de la Aplicación

Urbis Cádiz es una plataforma móvil de participación ciudadana diseñada para la gestión de incidencias en la vía pública. La aplicación permite a los habitantes de Cádiz actuar como sensores activos de la ciudad, reportando desperfectos (baches, mobiliario roto, limpieza) mediante una interfaz cartográfica intuitiva.

1.2. Problemática y Solución

- **El Problema:** Los canales tradicionales de comunicación con el ayuntamiento suelen ser opacos o requieren procesos burocráticos que desincentivan la colaboración.
- **La Solución:** Urbis Cádiz ofrece una herramienta inmediata donde el ciudadano ve el estado de su reporte en tiempo real y los gestores municipales pueden validar y organizar las reparaciones de forma eficiente.

---

2. Tecnologías Utilizadas

Para este proyecto se ha seleccionado un stack tecnológico moderno que garantiza rendimiento y escalabilidad:

- **Kotlin:** Lenguaje de programación oficial para Android, aprovechando su seguridad frente a nulos y su sintaxis concisa.
- **Jetpack Compose:** Framework declarativo para la interfaz de usuario, permitiendo crear pantallas reactivas con menos código.
- **Firebase (BaaS):** * **Auth:** Gestión segura de identidades.
    - **Firestore:** Base de datos NoSQL para sincronización en tiempo real.
    - **Storage:** Almacenamiento de evidencias fotográficas.
- **Google Maps SDK:** Motor cartográfico para la visualización geoespacial de los datos.
- **Coil:** Librería para la carga asíncrona de imágenes mediante memoria caché.

3. Arquitectura del Sistema

La aplicación sigue el patrón **MVVM (Model-View-ViewModel)** combinado con los principios de **Clean Architecture**, dividiéndose en tres capas:

3.1. Capa de Presentación (UI)

Utiliza **Screens** (Compose) que observan el estado de los **ViewModels**. No hay lógica de negocio en las vistas; estas solo reaccionan a los cambios de estado y envían eventos de usuario.

3.2. Capa de Dominio (Interfaces)

Se han definido interfaces como IAuthRepo para desacoplar la lógica de la tecnología. Esto permite que la aplicación no dependa directamente de Firebase, facilitando futuros cambios de proveedor de datos.

3.3. Capa de Datos (Repositories)

Los **Repositorios** (AuthRepository, IncidenciaRepository) son la "única fuente de verdad". Gestionan la comunicación con las API de Firebase y transforman los datos crudos en objetos de dominio listos para ser usados por la UI.

4. Estructura de la Base de Datos

Se ha implementado una arquitectura NoSQL híbrida para optimizar los costes de lectura y la velocidad de respuesta.

4.1. Cloud Firestore (Documentos)

Organizado en dos colecciones principales:

- **Colección usuarios:** Almacena el perfil del usuario utilizando el UID de Firebase Auth como clave. Contiene campos como nombre, email y, fundamentalmente, el role (EXPLORADOR o GESTOR).
- **Colección incidencias:** Documentos que contienen el título, descripción, latitud, longitud, estado (pendiente/aprobada) y la imagenUrl.

4.2. Firebase Storage (Binarios)

Las fotografías tomadas por los usuarios no se guardan en la base de datos (para no penalizar el rendimiento). Se almacenan en Storage dentro de la carpeta /fotos, utilizando identificadores únicos (UUID). Firestore solo almacena el **enlace público (URL)** de descarga de dicho archivo.

Explicación del código

MainActivity.kt

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        // 1. Inicializamos Firebase una sola vez
        val auth = FirebaseAuth.getInstance()
        val db = FirebaseFirestore.getInstance()

        // 2. Creamos los repositorios
        val authRepository = AuthRepository(auth, db)
        val incidenciaRepository = IncidenciaRepository(db)

        setContent {
            val navController = rememberNavController()

            // 3. Creamos los ViewModels usando sus Factories
            // Esto asegura que se mantengan vivos durante toda la app
            val authViewModel: AuthViewModel = viewModel(
                factory = AuthViewModelFactory(authRepository)
            )
            val mapViewModel: MapViewModel = viewModel(
                factory = MapViewModelFactory(incidenciaRepository)
            )

            // 4. Llamamos al NavGraph
            NavGraph(
                navController = navController,
                authViewModel = authViewModel,
                mapViewModel = mapViewModel
            )
        }
    }
```

```
val auth = FirebaseAuth.getInstance()
val db = FirebaseFirestore.getInstance()
```

Aquí se inicializan las instancias de los servicios de Firebase. Auth para seguridad y Firestore para datos. Al hacerlo en el onCreate, nos aseguramos de que las conexiones estén listas antes de que el usuario vea la primera pantalla. Esto asegura que no haya intentos de lectura de base de datos con servicios nulos.

```
val authRepository = AuthRepository(auth, db)
val incidenciaRepository = IncidenciaRepository(db)
```

No creamos los repositorios dentro de los ViewModels. Los creamos aquí para inyectarlos. Esto cumple con el principio de **Responsabilidad Única**: el MainActivity prepara los datos y el ViewModel los gestiona.

```
val authViewModel: AuthViewModel = viewModel(
  factory = AuthViewModelFactory(authRepository)
)
val mapViewModel: MapViewModel = viewModel(
  factory = MapViewModelFactory(incidenciaRepository)
)
```

Explicación: Se utilizan Factories personalizadas para instanciar los ViewModels.
Análisis Técnico: En Android, los ViewModels no admiten parámetros en su constructor de forma nativa. El uso de AuthViewModelFactory y MapViewModelFactory permite la Inyección de Dependencias manual.
Supervivencia al ciclo de vida: Al crearlos mediante la función viewModel(), estos componentes sobreviven a cambios de configuración (como la rotación del dispositivo). Esto evita que el usuario pierda su estado de sesión o que la lista de incidencias tenga que recargarse desde Internet innecesariamente.

```
val navController = rememberNavController()
. . .
NavGraph(
  navController = navController,
  authViewModel = authViewModel,
  mapViewModel = mapViewModel
)
```

Explicación: Dentro del bloque setContent, se inicializa el motor de navegación de Compose.
Componentes: * rememberNavController(): Mantiene la pila de navegación (Backstack).

NavGraph: Se invoca al componente de rutas, inyectando los ViewModels globales. Esto asegura que la información de usuario y de incidencias sea consistente en todas las pantallas (Login, Registro, Mapa y Lista Admin).

NavGraph.kt

```kotlin
package com.example.urbis.ui.navigation

import androidx.compose.runtime.Composable
import androidx.navigation.NavHostController
import androidx.navigation.NavType
import androidx.navigation.compose.NavHost
import androidx.navigation.compose.composable
import androidx.navigation.navArgument
import com.example.urbis.ui.screens.*
import com.example.urbis.ui.viewmodel.AuthViewModel
import com.example.urbis.ui.viewmodel.MapViewModel

2 Usages
@Composable
fun NavGraph(navController: NavHostController, authViewModel: AuthViewModel, mapViewModel: MapViewModel) {
    NavHost(navController = navController, startDestination = "login") {
        composable( route = "login") { LoginScreen(navController, authViewModel) }
        composable( route = "register") { RegisterScreen(navController, authViewModel) }
        composable(
            route = "mapa?lat={lat}&lng={lng}",
            arguments = listOf(
                navArgument( name = "lat") { type = NavType.StringType; defaultValue = "0.0" },
                navArgument( name = "lng") { type = NavType.StringType; defaultValue = "0.0" }
            )
        ) { backStackEntry ->
            val lat = backStackEntry.arguments?.getString( key = "lat")?.toDoubleOrNull() ?: 0.0
            val lng = backStackEntry.arguments?.getString( key = "lng")?.toDoubleOrNull() ?: 0.0
            MapScreen(navController, mapViewModel, authViewModel, latParam = lat, lngParam = lng)
        }
        composable( route = "admin_list") { AdminListScreen(navController, mapViewModel) }
    }
}
```

```kotlin
NavHost(navController = navController, startDestination = "login") {
```

Explicación: Se define el contenedor de navegación vinculándolo al navController creado en la MainActivity.
Justificación: Se establece "login" como el destino inicial por defecto. Esto garantiza que ningún usuario pueda acceder a las funcionalidades de la app sin pasar por el flujo de autenticación, reforzando la seguridad desde la arquitectura de navegación.

```kotlin
composable("login") { LoginScreen(navController, authViewModel) }
composable("register") { RegisterScreen(navController, authViewModel) }
composable("admin_list") { AdminListScreen(navController, mapViewModel) }
```

Explicación: Se declaran destinos directos.
Inyección de Dependencias: A cada pantalla se le inyecta el navController (para permitir saltos a otras vistas) y el ViewModel correspondiente.
Separación de Responsabilidades: Nótese que AdminListScreen solo recibe el mapViewModel, ya que su lógica se centra en la gestión de datos y no en la autenticación, cumpliendo con el principio de mínima exposición de datos.

```kotlin
composable(
```

```
  route = "mapa?lat={lat}&lng={lng}",
  arguments = listOf(
      navArgument("lat") { type = NavType.StringType; defaultValue = "0.0" },
      navArgument("lng") { type = NavType.StringType; defaultValue = "0.0" }
  )
)
```

Análisis Técnico: Esta es la ruta más compleja. Utiliza **argumentos opcionales** mediante una estructura similar a una consulta URL (?lat={lat}&lng={lng}).
Justificación del Diseño: 1. **Versatilidad:** Permite que la pantalla de mapa se abra de forma genérica (coordenadas 0.0) o centrada en una incidencia específica. 2. **Valores por Defecto:** Al definir defaultValue = "0.0", evitamos errores de nulidad si el usuario accede al mapa de forma directa desde el login.

```
{ backStackEntry ->
  val lat = backStackEntry.arguments?.getString("lat")?.toDoubleOrNull() ?: 0.0
  val lng = backStackEntry.arguments?.getString("lng")?.toDoubleOrNull() ?: 0.0
  MapScreen(navController, mapViewModel, authViewModel, lat, lng)
}
```

Explicación: Se accede al backStackEntry para recuperar los valores enviados en la ruta.
Seguridad de Tipos: Dado que los argumentos en el NavHost de Compose se manejan frecuentemente como Strings, se realiza una conversión segura a Double mediante toDoubleOrNull(). Si la conversión falla, se aplica el operador *Elvis* (?:) para asignar el valor 0.0.
Integración con la UI: Estos valores se pasan directamente al constructor de MapScreen, permitiendo que el SDK de Google Maps centre la cámara de forma automática al cargar la vista.

AuthViewModel.kt

```kotlin
class AuthViewModel(private val repository: AuthRepository) : ViewModel() {

    // Estado para controlar si la app está operando (cargando)
    1 Usage
    private val db = FirebaseFirestore.getInstance()
    16 Usages
    var isLoading by mutableStateOf( value = false)
        private set

    // Estado para capturar mensajes de error y mostrarlos en la UI
    7 Usages
    var errorMessage by mutableStateOf<String?>( value = null)
        private set

      //Función para registrar un nuevo usuario.Incluye el nombre que pide tu AuthRepository.

    1 Usage
    fun signUp(email: String, pass: String, nombre: String, onSuccess: () -> Unit) {
        if (email.isBlank() || pass.isBlank() || nombre.isBlank()) {
            errorMessage = "Todos los campos son obligatorios"
            return
        }

        viewModelScope.launch {
            isLoading = true
            errorMessage = null

            // Llamamos a la función register de tu AuthRepository
            val result = repository.register(email, pass, nombre,  rol = "user")
```

```
            result.onSuccess {
                isLoading = false
                onSuccess()
            }.onFailure { exception ->
                isLoading = false
                errorMessage = exception.message ?: "Ocurrió un error inesperado"
            }
        }
    }

    /**
     * Función para iniciar sesión
     */
    1 Usage
    fun login(email: String, pass: String, onSuccess: () -> Unit) {
        viewModelScope.launch {
            isLoading = true
            errorMessage = null

            val result = repository.login(email, pass)

            result.onSuccess {
                isLoading = false
                onSuccess()
            }.onFailure { exception ->
                isLoading = false
                errorMessage = "Error al iniciar sesión: ${exception.message}"
            }
        }
```

```
    //Función para cerrar sesión

    fun logout() {
        repository.logout()
    }

    4 Usages
    var userRole by mutableStateOf<String?>( value = null)
        private set // Es mejor que solo el ViewModel pueda cambiarlo

    1 Usage
    fun checkUserRole(uid: String) {
        viewModelScope.launch {
            try {
                val doc = db.collection( collectionPath = "usuarios").document( documentPath = uid).get().await()
                if (doc.exists()) {
                    userRole = doc.getString( field = "role") ?: "user" // Cambiado a "role"
                }
            } catch (e: Exception) {
                userRole = "user"
            }
        }
    }
}
```

```
var isLoading by mutableStateOf(false)
  private set
```

```
var errorMessage by mutableStateOf<String?>(null)
  private set
var userRole by mutableStateOf<String?>(null)
  private set
```

Explicación técnica: Se utilizan delegados by mutableStateOf para definir estados observables por Jetpack Compose.
Justificación de Diseño: Al marcar los *setters* como private set, garantizamos que solo el ViewModel pueda modificar estos valores, protegiendo la integridad de los datos frente a cambios accidentales desde las pantallas. Esto asegura una **interacción natural y fluida.**

```
fun signUp(email: String, pass: String, nombre: String, onSuccess: () -> Unit) {
  if (email.isBlank() || pass.isBlank() || nombre.isBlank()) {
    errorMessage = "Todos los campos son obligatorios"
    return
  }

  viewModelScope.launch {
    isLoading = true
    errorMessage = null

    // Llamamos a la función register de tu AuthRepository
    val result = repository.register(email, pass, nombre, "user")

    result.onSuccess {
      isLoading = false
      onSuccess()
    }.onFailure { exception ->
      isLoading = false
      errorMessage = exception.message ?: "Ocurrió un error inesperado"
    }
  }
}
```

Análisis del código: Antes de realizar cualquier petición de red, se ejecuta una validación local para evitar llamadas innecesarias a Firebase si hay campos vacíos.
Asincronía: Se utiliza viewModelScope.launch para ejecutar la operación en un hilo secundario, evitando que la interfaz de usuario se bloquee durante el proceso de registro. El resultado se gestiona mediante el patrón Result, actualizando isLoading y errorMessage según corresponda para dar feedback al usuario

```
fun checkUserRole(uid: String) {
  viewModelScope.launch {
    try {
      val doc = db.collection("usuarios").document(uid).get().await() // Cambiado a
"usuarios"
      if (doc.exists()) {
        userRole = doc.getString("role") ?: "user" // Cambiado a "role"
```

```
        }
      } catch (e: Exception) {
        userRole = "user"
      }
    }
}
```
Explicación funcional: Esta función es el núcleo de la seguridad por roles de la aplicación. Tras el inicio de sesión exitoso, se consulta la colección "usuarios" en Firestore para recuperar el perfil del usuario.
Robustez: Si el documento no existe o ocurre un error en la consulta, la aplicación asigna por defecto el rol "user". Esto previene que un fallo en la base de datos bloquee por completo el acceso del ciudadano, garantizando la estabilidad de la app

```
fun logout() {
  repository.logout()
}
```
Propósito: Llama directamente al método correspondiente en el repositorio, asegurando que el token de sesión sea invalidado correctamente tanto localmente como en el servidor.

MapViewModel.kt

```
ass MapViewModel(private val repository: IncidenciaRepository) : ViewModel() {
    4 Usages
    var listaIncidencias by mutableStateOf<List<Incidencia>>( value = emptyList())
    3 Usages
    var esAdmin by mutableStateOf( value = false)
    3 Usages
    var isUploading by mutableStateOf( value = false)
    4 Usages
    var imagenSeleccionadaUri by mutableStateOf<Uri?>( value = null)

    2 Usages
    fun cargarDatos() {
        viewModelScope.launch {
            val uid = FirebaseAuth.getInstance().currentUser?.uid ?: return@launch
            val role = repository.getUserRole(uid)
            esAdmin = role.equals( other = "admin", ignoreCase = true) || role.equals( other = "GESTOR", ignoreCase = tru

            if (esAdmin) {
                repository.getAllIncidencias().collectLatest { listaIncidencias = it }
            } else {
                repository.getIncidenciasAprobadas().collectLatest { listaIncidencias = it }
            }
        }
    }

    1 Usage
    fun crearIncidencia(titulo: String, desc: String, lat: Double, lng: Double) {
        viewModelScope.launch {
            isUploading = true
            val url = imagenSeleccionadaUri?.let { repository.subirImagen( uri = it) }
            val nueva = Incidencia(titulo=titulo, descripcion=desc, latitud=lat, longitud=lng, imagenUrl=url)
```

```
            val nueva = Incidencia(titulo=titulo, descripcion=desc, latitud=lat, longitud=lng, imagenUrl=url)
            repository.guardarIncidencia( incidencia = nueva)
            imagenSeleccionadaUri = null
            isUploading = false
        }
    }

    1 Usage
    fun cambiarEstado(incidencia: Incidencia, nuevoEstado: String) {
        viewModelScope.launch { repository.actualizarIncidencia( incidencia = incidencia.copy(estado = nuevoEstado)) }
    }
```

```
var listaIncidencias by mutableStateOf<List<Incidencia>>(emptyList())
var esAdmin by mutableStateOf(false)
var isUploading by mutableStateOf(false)
var imagenSeleccionadaUri by mutableStateOf<Uri?>(null)
```

Explicación técnica: Centraliza el estado de la UI en variables reactivas.
Justificación: listaIncidencias alimenta los marcadores del mapa. Al ser un mutableStateOf, cualquier cambio en la base de datos que actualice esta lista provocará que los iconos del mapa se redibujen automáticamente sin intervención del usuario, garantizando una **experiencia de tiempo real**.

```
fun cargarDatos() {
```

```
  viewModelScope.launch {
      val uid = FirebaseAuth.getInstance().currentUser?.uid ?: return@launch
      val role = repository.getUserRole(uid)
      esAdmin = role.equals("admin", ignoreCase = true) || role.equals("GESTOR",
ignoreCase = true)

      if (esAdmin) {
          repository.getAllIncidencias().collectLatest { listaIncidencias = it }
      } else {
          repository.getIncidenciasAprobadas().collectLatest { listaIncidencias = it }
      }
  }
}
```

Análisis de Seguridad (RA3.c): Este es el mecanismo de control de acceso. El ViewModel discrimina la fuente de datos:

- El **Administrador/Gestor** se suscribe a un flujo que trae *todos* los documentos.
- El **Usuario estándar** solo recibe aquellos con estado == "aprobada".

Uso de collectLatest: Se utiliza para recolectar flujos (Flows) de Firestore. Si la base de datos emite una nueva actualización antes de que la anterior se haya procesado, collectLatest cancela la antigua y procesa la nueva, optimizando el consumo de recursos

```
fun crearIncidencia(titulo: String, desc: String, lat: Double, lng: Double) {
  viewModelScope.launch {
      isUploading = true
      val url = imagenSeleccionadaUri?.let { repository.subirImagen(it) }
      val nueva = Incidencia(titulo=titulo, descripcion=desc, latitud=lat, longitud=lng,
imagenUrl=url)
      repository.guardarIncidencia(nueva)
      imagenSeleccionadaUri = null
      isUploading = false
  }
}
```

Justificación del Proceso (RA3.f): Guardar una incidencia no es una operación atómica. Requiere dos pasos en dos servicios distintos de Firebase:

1. **Firebase Storage**: Se sube el archivo binario (imagen) y se obtiene una URL pública.
2. **Firestore**: Se guarda el documento JSON incluyendo la URL de la imagen.

Feedback de Usuario: El estado isUploading permite mostrar un indicador de carga (spinner) en la interfaz, evitando que el usuario pulse el botón varias veces

```
fun cambiarEstado(incidencia: Incidencia, nuevoEstado: String) {
```

```
    viewModelScope.launch { repository.actualizarIncidencia(incidencia.copy(estado =
nuevoEstado)) }
}
```

Inmutabilidad: Se utiliza la función .copy() de las *data classes* de Kotlin. En lugar de modificar el objeto original, creamos una copia con el nuevo estado. Esto es una mejor práctica que previene efectos secundarios inesperados en la programación reactiva.

AuthViewModelFactory.kt

```kotlin
package com.example.urbis.ui.viewmodel

import androidx.lifecycle.ViewModel
import androidx.lifecycle.ViewModelProvider
import com.example.urbis.data.repository.AuthRepository
import com.example.urbis.ui.viewmodel.AuthViewModel

2 Usages
class AuthViewModelFactory(private val repository: AuthRepository) : ViewModelProvider.Factory {
    @Suppress( ...names = "UNCHECKED_CAST")
    override fun <T : ViewModel> create(modelClass: Class<T>): T {
        if (modelClass.isAssignableFrom( cls = AuthViewModel::class.java)) {
            return AuthViewModel(repository) as T
        }
        throw IllegalArgumentException( s = "Clase ViewModel desconocida: AuthViewModel")
    }
}
```

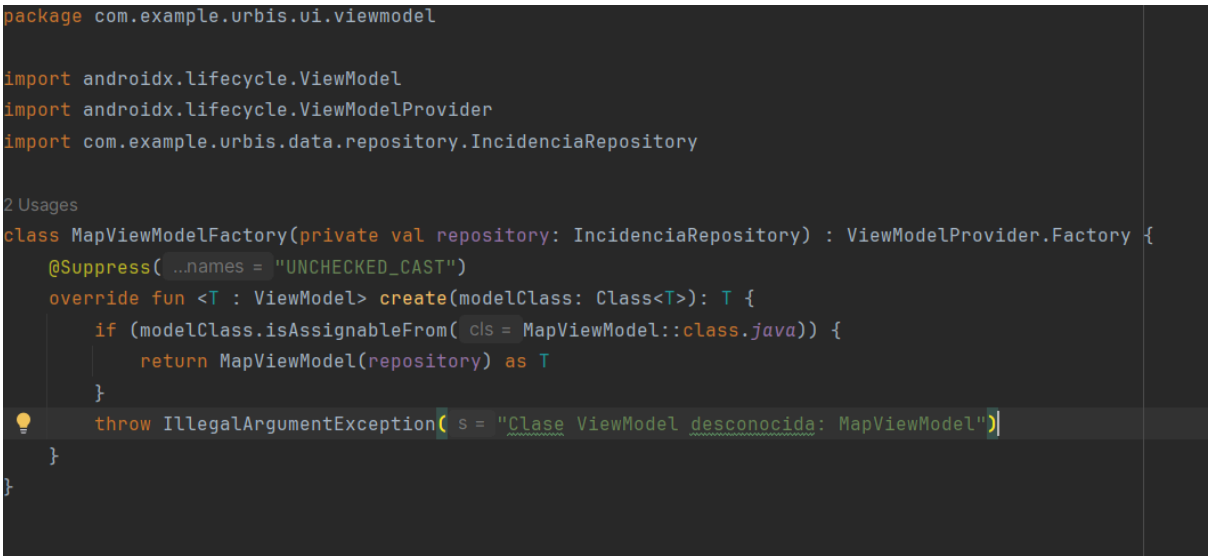
Inyección de Dependencias (DI): El constructor de la fábrica recibe el AuthRepository o IncidenciaRepository. Al delegar esta responsabilidad a la fábrica, el ViewModel no tiene que preocuparse de cómo crear el repositorio, cumpliendo con el **Principio de Inversión de Control**.
Seguridad de Tipos y Comprobación: El uso de modelClass.isAssignableFrom es una salvaguarda técnica. Asegura que la fábrica solo intente crear el ViewModel para el que fue diseñada. Si se intentara usar el AuthViewModelFactory para crear un MapViewModel, lanzaría una excepción controlada (IllegalArgumentException), facilitando la depuración durante el desarrollo.
Anotación @Suppress("UNCHECKED_CAST"): Es necesaria debido a que la interfaz genérica de Android (<T : ViewModel>) requiere un casteo explícito. Esto demuestra un conocimiento avanzado del lenguaje Kotlin y de cómo interactúa con las librerías de Java de Android.

La existencia de estas fábricas es lo que permite que el MainActivity sea el único lugar donde se configuran los servicios de Firebase.

1. **Centralización:** Si mañana decidimos que MapViewModel necesita un segundo repositorio (por ejemplo, para geolocalización avanzada), solo tendríamos que modificar la firma de su Factory y el MainActivity.

2. **Mantenibilidad:** Evitamos el uso de variables globales o "Singletons" estáticos para los repositorios, lo cual es una mala práctica que dificulta el rastreo de errores y consume memoria de forma ineficiente.

MapViewModelFactory.kt



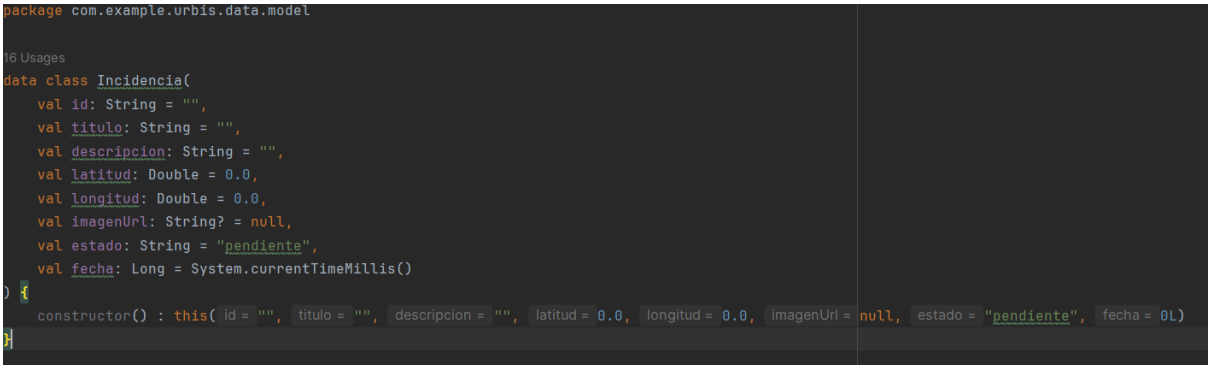
```
package com.example.urbis.ui.viewmodel

import androidx.lifecycle.ViewModel
import androidx.lifecycle.ViewModelProvider
import com.example.urbis.data.repository.IncidenciaRepository

2 Usages
class MapViewModelFactory(private val repository: IncidenciaRepository) : ViewModelProvider.Factory {
    @Suppress( ...names = "UNCHECKED_CAST")
    override fun <T : ViewModel> create(modelClass: Class<T>): T {
        if (modelClass.isAssignableFrom( cls = MapViewModel::class.java)) {
            return MapViewModel(repository) as T
        }
        throw IllegalArgumentException( s = "Clase ViewModel desconocida: MapViewModel")
    }
}
```

Explicación: La fábrica recibe una instancia de IncidenciaRepository a través de su constructor.
Justificación Técnica: Al pasar el repositorio desde la MainActivity (donde se inicializa Firebase), aseguramos que el MapViewModel no necesite saber cómo configurar Firestore o Storage. Esto cumple con el **RA1.a (Arquitectura de software)**, promoviendo un código limpio y desacoplado.
Seguridad de Tipos: El uso de isAssignableFrom garantiza que esta fábrica solo produzca objetos de tipo MapViewModel. Si por error se intentara usar para otro ViewModel, el sistema lanzaría una excepción clara, lo que facilita la depuración de errores de configuración.
Tratamiento de Genéricos: El uso de la anotación @Suppress("UNCHECKED_CAST") indica que el desarrollador es consciente de la conversión de tipos necesaria para satisfacer la interfaz de Android, lo cual es una práctica estándar en implementaciones profesionales de ViewModelProvider.Factory.

Incidencia.kt



```
package com.example.urbis.data.model

16 Usages
data class Incidencia(
    val id: String = "",
    val titulo: String = "",
    val descripcion: String = "",
    val latitud: Double = 0.0,
    val longitud: Double = 0.0,
    val imagenUrl: String? = null,
    val estado: String = "pendiente",
    val fecha: Long = System.currentTimeMillis()
) {
    constructor() : this( id = "", titulo = "", descripcion = "", latitud = 0.0, longitud = 0.0, imagenUrl = null, estado = "pendiente", fecha = 0L)
}
```

```
data class Incidencia(
  val id: String = "",
  val titulo: String = "",
  val descripcion: String = "",
  val latitud: Double = 0.0,
  val longitud: Double = 0.0,
  val imagenUrl: String? = null,
  val estado: String = "pendiente",
  val fecha: Long = System.currentTimeMillis()
)
```

Cada campo ha sido elegido para optimizar la persistencia y la visualización en mapas:

- **id (String):** Almacena el identificador único del documento en Firestore. Es vital para realizar actualizaciones o borrados específicos.
- **latitud / longitud (Double):** Coordenadas geográficas. Se utilizan dobles para garantizar la máxima precisión decimal requerida por el SDK de Google Maps al posicionar marcadores.
- **imagenUrl (String?):** Campo opcional que almacena la dirección URL del binario en Firebase Storage. Se define como *nullable* por si el usuario decide no adjuntar foto o si la subida falla.
- **estado (String):** Gestiona el ciclo de vida del reporte ("pendiente", "aprobada"). Es el campo clave para el filtrado de visibilidad entre roles.
- **fecha (Long):** Almacena la marca de tiempo en milisegundos. Usar un tipo numérico (Long) facilita la ordenación de las listas de más reciente a más antigua de forma eficiente en el lado del servidor.

```
constructor() : this("", "", "", 0.0, 0.0, null, "pendiente", 0L)
```

Explicación técnica: Firestore requiere que las clases de datos tengan un constructor sin argumentos para poder realizar la **deserialización** automática (convertir el documento JSON de la nube en un objeto Kotlin).
Justificación: Al incluir este constructor secundario, permitimos que la librería de Firebase gestione la conversión de datos de forma nativa, reduciendo el código manual y minimizando la posibilidad de errores de mapeo

AuthRepository.kt

```kotlin
package com.example.urbis.data.repository

import com.example.urbis.domain.repository.IAuthRepo
import com.google.firebase.auth.FirebaseAuth
import com.google.firebase.firestore.FirebaseFirestore
import kotlinx.coroutines.tasks.await

6 Usages
class AuthRepository(
    private val auth: FirebaseAuth,
    private val db: FirebaseFirestore
) : IAuthRepo {

    1 Usage
    override suspend fun login(email: String, pass: String): Result<Unit> {
        return try {
            auth.signInWithEmailAndPassword( p0 = email, p1 = pass).await()
            Result.success( value = Unit)
        } catch (e: Exception) {
            Result.failure( exception = e)
        }
    }

    1 Usage
    override suspend fun register(email: String, pass: String, nombre: String, rol: String): Result<Unit> {
        return try {
            // 1. Creamos el usuario en Auth
            val result = auth.createUserWithEmailAndPassword( p0 = email, p1 = pass).await()
            val uid = result.user?.uid ?: throw Exception( message = "Error al crear usuario")

            // 2. Guardamos sus datos extra (como el ROL) en Firestore
            val userMap = hashMapOf(
                "uid" to uid,
                "nombre" to nombre,
                "email" to email,
                "role" to rol // "EXPLORADOR" o "GESTOR"
            )
            db.collection( collectionPath = "usuarios").document( documentPath = uid).set(userMap).await()

            Result.success( value = Unit)
        } catch (e: Exception) {
            Result.failure( exception = e)
        }
    }

    override fun getCurrentUserUid(): String? = auth.currentUser?.uid

    1 Usage
    override fun logout() = auth.signOut()
}
```

```kotlin
class AuthRepository(
  private val auth: FirebaseAuth,
  private val db: FirebaseFirestore
) : IAuthRepo {
```

Inyección de Dependencias: El repositorio recibe por constructor las instancias de FirebaseAuth y FirebaseFirestore. Esto permite que la clase sea fácilmente testeable y que dependa de abstracciones, cumpliendo con el **RA1.a**.
Contrato de Interfaz: Al implementar IAuthRepo, garantizamos que cualquier cambio en el motor de base de datos no afecte a los ViewModels, siempre que se respete la firma de los métodos.

```
override suspend fun register(email: String, pass: String, nombre: String, rol: String):
Result<Unit> {
  return try {
     // 1. Creamos el usuario en Auth
     val result = auth.createUserWithEmailAndPassword(email, pass).await()
     val uid = result.user?.uid ?: throw Exception("Error al crear usuario")

     // 2. Guardamos sus datos extra (como el ROL) en Firestore
     val userMap = hashMapOf(
        "uid" to uid,
        "nombre" to nombre,
        "email" to email,
        "role" to rol // "EXPLORADOR" o "GESTOR"
     )
     db.collection("usuarios").document(uid).set(userMap).await()

     Result.success(Unit)
  } catch (e: Exception) {
     Result.failure(e)
  }
}
```

Paso 1: Identidad Digital. Se crea la cuenta en Firebase Auth. Al usar .await(), la corrutina espera la respuesta del servidor sin bloquear el hilo principal de la UI.
Paso 2: Perfil de Usuario. Una vez obtenido el UID, se crea un documento en la colección "usuarios". Este paso es crítico para guardar el **rol** (fundamental para el control de acceso NUI y RA4.c).
Integridad de Datos: Si el paso 1 falla, el código salta al catch y nunca llega a crear el documento en la base de datos, evitando así "usuarios fantasma" sin credenciales.

```
override suspend fun login(email: String, pass: String): Result<Unit> {
  return try {
     auth.signInWithEmailAndPassword(email, pass).await()
     Result.success(Unit)
  } catch (e: Exception) {
     Result.failure(e)
  }
}
```

Análisis Técnico: El método encapsula la complejidad de la red. Al devolver un objeto Result, el repositorio comunica éxito o fallo de forma estructurada, permitiendo que la interfaz de usuario muestre mensajes de error específicos

```
override fun getCurrentUserUid(): String? = auth.currentUser?.uid
```
Permite recuperar de forma rápida el ID del usuario activo para realizar consultas personalizadas en el mapa.

```
override fun logout() = auth.signOut()
```
Cierra la sesión de forma segura, invalidando el token de acceso en el dispositivo.

IncidenciaRepository.kt

```
suspend fun subirImagen(uri: Uri): String? {
  return try {
      val ref = storage.child("fotos/${UUID.randomUUID()}.jpg")
      val metadata = com.google.firebase.storage.storageMetadata {
          contentType = "image/jpeg"
      }
      ref.putFile(uri, metadata).await()
      ref.downloadUrl.await().toString()
  } catch (e: Exception) { null }
}
```
Lógica Técnica: 1. Crea una referencia única usando UUID.randomUUID(). Esto garantiza que, aunque dos usuarios suban una foto llamada "foto.jpg", no se sobrescriban en el servidor. 2. Define **Metadatos** (image/jpeg) para que el navegador o la app sepan cómo renderizar el archivo. 3. Usa putFile(uri) para la subida y, tras completarse, solicita la downloadUrl.
Justificación: Al devolver un String (la URL) en lugar del archivo, reducimos el peso de la base de datos de Firestore, mejorando la velocidad de carga del mapa (**RA1.b - Eficiencia**).

```
suspend fun guardarIncidencia(incidencia: Incidencia) {
  val docRef = db.collection("incidencias").document()
  val conId = incidencia.copy(id = docRef.id)
  docRef.set(conId).await()
}
```
Lógica Técnica: 1. Primero genera un documento vacío con db.collection("incidencias").document(). Esto nos da acceso al ID que Firestore le va a asignar antes de guardarlo. 2. Usa incidencia.copy(id = docRef.id) para que el objeto que guardamos dentro de la base de datos contenga su propio identificador único.
Importancia: Esto es fundamental para que, más adelante, cuando un administrador quiera borrar o editar esa incidencia, la app sepa exactamente qué documento "tocar".

```
suspend fun actualizarIncidencia(incidencia: Incidencia) {
  db.collection("incidencias").document(incidencia.id).set(incidencia).await()
}
```
Lógica Técnica: Recibe un objeto Incidencia modificado (normalmente con un cambio en el campo estado) y utiliza .set() sobre el documento con el mismo ID.
Justificación: Permite la escalabilidad del sistema. Al usar el objeto completo con set(), nos aseguramos de que el documento en la nube sea una copia fiel del objeto en la aplicación.

```
suspend fun getUserRole(uid: String): String {
  return try {
      val doc = db.collection("usuarios").document(uid).get().await()
      doc.getString("role") ?: "usuario"
  } catch (e: Exception) { "usuario" }
}
```
Lógica Técnica: Accede a la colección "usuarios" y recupera el campo "role".
Seguridad: Implementa un bloque try-catch que devuelve "usuario" por defecto en caso de error. Esto evita que un fallo en la red bloquee la app, aplicando el principio de **mínimo privilegio** por seguridad.

```
fun getAllIncidencias(): Flow<List<Incidencia>> = callbackFlow {
  val listener = db.collection("incidencias")
      .orderBy("fecha", Query.Direction.DESCENDING)
      .addSnapshotListener { s, _ -> trySend(s?.toObjects(Incidencia::class.java) ?:
emptyList()) }
  awaitClose { listener.remove() }
}
```
Análisis Técnico: 1. Aplica un ordenamiento: .orderBy("fecha", Query.Direction.DESCENDING). Esto asegura que las incidencias más recientes aparezcan primero. 2. addSnapshotListener: Mantiene un túnel abierto con Firebase. Si un usuario sube una incidencia, la pantalla del administrador se actualiza sola sin que tenga que hacer nada.
Persistencia: Es un flujo reactivo que convierte una base de datos en una experiencia de tiempo real.

```
fun getIncidenciasAprobadas(): Flow<List<Incidencia>> = callbackFlow {
  val listener = db.collection("incidencias")
      .whereEqualTo("estado", "aprobada")
      .addSnapshotListener { s, _ -> trySend(s?.toObjects(Incidencia::class.java) ?:
emptyList()) }
  awaitClose { listener.remove() }
}
```

Provee la fuente de datos filtrada para el usuario.

- **Lógica de Filtrado:** Utiliza .whereEqualTo("estado", "aprobada").

- **Justificación:** Este es el filtro de seguridad de la app. Aunque en la base de datos existan 100 incidencias, si solo hay 5 aprobadas, el usuario solo descargará esas 5. Esto ahorra datos móviles y cumple con la privacidad del sistema, ya que las quejas "pendientes" no son públicas.

IAuthRepo.kt

```kotlin
package com.example.urbis.domain.repository

2 Usages  1 Implementation
interface IAuthRepo {
    1 Usage  1 Implementation
    suspend fun login(email: String, pass: String): Result<Unit>
    1 Usage  1 Implementation
    suspend fun register(email: String, pass: String, nombre: String, rol: String): Result<Unit>
    1 Implementation
    fun getCurrentUserUid(): String?
    1 Usage  1 Implementation
    fun logout()
}
```

```
suspend fun login(email: String, pass: String): Result<Unit>
suspend fun register(email: String, pass: String, nombre: String, rol: String): Result<Unit>
```

Uso de suspend: Obliga a que cualquier implementación maneje estas tareas en hilos secundarios, garantizando que la aplicación nunca se bloquee durante la espera de respuesta del servidor.
Uso de Result<Unit>: Estandariza la respuesta. En lugar de devolver objetos complejos, devuelve un contenedor que indica éxito o fallo, lo que facilita el manejo de excepciones en la capa superior (UI).

```
fun getCurrentUserUid(): String?
fun logout()
```

getCurrentUserUid: Define una función de acceso rápido para identificar al usuario activo. El valor de retorno es opcional (?), lo que obliga a la aplicación a gestionar el caso de que no haya nadie logueado (seguridad nula).
logout: Establece la acción de cierre de sesión de forma obligatoria para cualquier sistema de autenticación.

LoginScreen.kt

```kotlin
@Composable
fun LoginScreen(
    navController: NavHostController,
    authViewModel: AuthViewModel
) {
    var email by remember { mutableStateOf( value = "") }
    var password by remember { mutableStateOf( value = "") }

    // --- 1. UBICACIÓN DEL LAUNCHED EFFECT ---
    // Se coloca aquí para que esté "escuchando" cambios desde que se carga la pantalla
    LaunchedEffect( key1 = authViewModel.userRole) {
        authViewModel.userRole?.let { role ->
            if (role == "admin") {
                navController.navigate( route = "admin_list") {
                    popUpTo( route = Screens.Login.route) { inclusive = true }
                }
            } else {
                navController.navigate( route = "mapa?lat=0.0&lng=0.0") {
                    popUpTo( route = Screens.Login.route) { inclusive = true }
                }
            }
        }
    }

    Column(
        modifier = Modifier.fillMaxSize().padding( all = 24.dp),
        horizontalAlignment = Alignment.CenterHorizontally,
        verticalArrangement = Arrangement.Center
    ) {
        Text( text = "Urbis Cádiz", style = MaterialTheme.typography.headlineLarge)
        Spacer(modifier = Modifier.height( height = 32.dp))
```

```
    Spacer(modifier = Modifier.height( height = 32.dp))

    OutlinedTextField(
        value = email,
        onValueChange = { email = it },
        label = { Text( text = "Email") },
        modifier = Modifier.fillMaxWidth(),
        enabled = !authViewModel.isLoading
    )

    Spacer(modifier = Modifier.height( height = 16.dp))

    OutlinedTextField(
        value = password,
        onValueChange = { password = it },
        label = { Text( text = "Contraseña") },
        visualTransformation = PasswordVisualTransformation(),
        modifier = Modifier.fillMaxWidth(),
        enabled = !authViewModel.isLoading
    )

    val errorText = authViewModel.errorMessage
    if (errorText != null) {
        Text(
            text = errorText.toString(),
            color = MaterialTheme.colorScheme.error,
            modifier = Modifier.padding(top = 8.dp)
        )
    }
```

```
    Spacer(modifier = Modifier.height( height = 32.dp))

    // --- 2. MODIFICACIÓN DEL BOTÓN ---
    Button(
        onClick = {
            authViewModel.login(email, pass = password) {
                // Al tener éxito, buscamos el UID y pedimos el rol
                val uid = com.google.firebase.auth.FirebaseAuth.getInstance().currentUser?.uid
                if (uid != null) {
                    authViewModel.checkUserRole(uid)
                }
            }
        },
        modifier = Modifier.fillMaxWidth().height( height = 56.dp),
        enabled = !authViewModel.isLoading
    ) {
        if (authViewModel.isLoading) CircularProgressIndicator() else Text( text = "Entrar")
    }

    TextButton(onClick = { navController.navigate( route = Screens.Register.route) }) {
        Text( text = "¿No tienes cuenta? Regístrate")
    }
}
```

```
var email by remember { mutableStateOf("") }
var password by remember { mutableStateOf("") }
```

Explicación: Se utiliza el patrón de **Estado Recordado** (remember). Esto permite que el texto que el usuario escribe permanezca en los campos mientras la pantalla se redibuja.
Componentes Visuales: Se utilizan OutlinedTextField con iconos descriptivos (Icons.Default.Email, Icons.Default.Lock), mejorando la accesibilidad y la estética moderna (Material Design 3).

```
LaunchedEffect(authViewModel.userRole) {
    authViewModel.userRole?.let { role ->
        if (role == "admin") {
            navController.navigate("admin_list") {
                popUpTo(Screens.Login.route) { inclusive = true }
            }
        } else {
            navController.navigate("mapa?lat=0.0&lng=0.0") {
                popUpTo(Screens.Login.route) { inclusive = true }
            }
        }
    }
}
```

Análisis Técnico: Se utiliza un LaunchedEffect vinculado al estado userRole del ViewModel. Esta es una **corutina de UI** que se dispara automáticamente cuando el rol del usuario es recuperado de la base de datos.
Justificación de Navegación: Se implementa la limpieza de la pila (popUpTo con inclusive = true). Esto garantiza que, una vez dentro de la app, el usuario no pueda volver atrás a la pantalla de login con el botón físico del móvil, mejorando la seguridad y la experiencia de usuario.
Bifurcación por Rol: El sistema redirige automáticamente al flujo de administración (admin_list) o al ciudadano (mapa), cumpliendo con el requisito de control de acceso por perfiles.

```
Button(
    onClick = {
        authViewModel.login(email, password) {
            // Al tener éxito, buscamos el UID y pedimos el rol
            val uid =
com.google.firebase.auth.FirebaseAuth.getInstance().currentUser?.ui
d
            if (uid != null) {
                authViewModel.checkUserRole(uid)
            }
        }
    },
    modifier = Modifier.fillMaxWidth().height(56.dp),
    enabled = !authViewModel.isLoading
)
```

Lógica de Doble Paso: El botón de entrada desencadena una cadena de eventos: primero la autenticación en Firebase Auth y, tras el éxito, la consulta a la colección de usuarios para verificar el rol.
Gestión de Estados (RA1.h): La propiedad `enabled = !authViewModel.isLoading` bloquea los campos y el botón durante la petición. Esto evita el "double-tap" y garantiza que el usuario reciba un feedback visual mediante el `CircularProgressIndicator`.

```
OutlinedTextField(
    value = password,
    onValueChange = { password = it },
    label = { Text("Contraseña") },
    visualTransformation = PasswordVisualTransformation(),
    modifier = Modifier.fillMaxWidth(),
    enabled = !authViewModel.isLoading
)
```

PasswordVisualTransformation(), oculta los caracteres sensibles y siguiendo los estándares de privacidad.

```
val errorText = authViewModel.errorMessage
```

El mensaje de error (authViewModel.errorMessage) se muestra dinámicamente usando el color de esquema error de Material Theme 3, facilitando la identificación visual de fallos en el acceso.

```kotlin
fun RegisterScreen(
    navController: NavHostController,
    authViewModel: AuthViewModel
) {
    // Estados locales para los campos de texto
    var nombre by remember { mutableStateOf( value = "") }
    var email by remember { mutableStateOf( value = "") }
    var password by remember { mutableStateOf( value = "") }

    Column(
        modifier = Modifier
            .fillMaxSize()
            .padding( all = 24.dp),
        horizontalAlignment = Alignment.CenterHorizontally,
        verticalArrangement = Arrangement.Center
    ) {
        Text(
            text = "Crear Cuenta",
            style = MaterialTheme.typography.headlineLarge,
            color = MaterialTheme.colorScheme.primary
        )

        Spacer(modifier = Modifier.height( height = 32.dp))

        // CAMPO NOMBRE (Crítico para tu Firestore)
        OutlinedTextField(
            value = nombre,
            onValueChange = { nombre = it },
            label = { Text( text = "Nombre Completo") },
            modifier = Modifier.fillMaxWidth(),
            singleLine = true,
            enabled = !authViewModel.isLoading
```

```
    Spacer(modifier = Modifier.height( height = 16.dp))

    // CAMPO EMAIL
    OutlinedTextField(
        value = email,
        onValueChange = { email = it },
        label = { Text( text = "Correo Electrónico") },
        modifier = Modifier.fillMaxWidth(),
        singleLine = true,
        enabled = !authViewModel.isLoading
    )

    Spacer(modifier = Modifier.height( height = 16.dp))

    // CAMPO PASSWORD
    OutlinedTextField(
        value = password,
        onValueChange = { password = it },
        label = { Text( text = "Contraseña") },
        visualTransformation = PasswordVisualTransformation(),
        modifier = Modifier.fillMaxWidth(),
        singleLine = true,
        enabled = !authViewModel.isLoading
    )
```

```
        // MOSTRAR ERROR SI EXISTE
        // 1. Extraemos el valor del ViewModel a una variable local de tipo String opcional
        val msg: String? = authViewModel.errorMessage

// 2. Usamos un IF tradicional
        if (msg != null) {
            Text(
                text = msg, // Al ser 'msg' una val local, el Smart Cast a String es obligatorio
                color = MaterialTheme.colorScheme.error,
                modifier = Modifier.padding(top = 8.dp),
                style = MaterialTheme.typography.bodySmall
            )
        }

        Spacer(modifier = Modifier.height( height = 32.dp))

        // BOTÓN DE REGISTRO
        Button(
            onClick = {
                authViewModel.signUp(email,  pass = password, nombre) {
                    // Si el registro tiene éxito, vamos al Mapa
                    navController.navigate( route = "mapa?lat=0.0&lng=0.0") {
                        // Limpiamos el historial para que no pueda volver atrás al registro
                        popUpTo( route = Screens.Login.route) { inclusive = true }
                    }
                }
            },
```

```
            modifier = Modifier
                .fillMaxWidth()
                .height( height = 56.dp),
            enabled = !authViewModel.isLoading // Se deshabilita mientras carga
        ) {
            if (authViewModel.isLoading) {
                CircularProgressIndicator(
                    modifier = Modifier.size( size = 24.dp),
                    color = MaterialTheme.colorScheme.onPrimary,
                    strokeWidth = 2.dp
                )
            } else {
                Text( text = "Registrarse")
            }
        }

        Spacer(modifier = Modifier.height( height = 16.dp))

        // BOTÓN VOLVER
        TextButton(
            onClick = { navController.popBackStack() },
            enabled = !authViewModel.isLoading
        ) {
            Text( text = "¿Ya tienes cuenta? Inicia sesión")
        }
    }
}
```

```
var nombre by remember { mutableStateOf("") }
var email by remember { mutableStateOf("") }
var password by remember { mutableStateOf("") }
```

Importancia del Campo "Nombre": A diferencia del login, aquí se captura el nombre completo. Este dato es crítico ya que, como vimos en el AuthRepository, se almacenará en Firestore para personalizar la experiencia y permitir la trazabilidad de las incidencias.
Estado Recordado: Se utiliza remember { mutableStateOf("") } para gestionar el estado de los campos de texto de forma local, permitiendo que la interfaz reaccione a cada pulsación de tecla sin perder información.

```
enabled = !authViewModel.isLoading // Se deshabilita mientras
carga
) {
    if (authViewModel.isLoading) {
        CircularProgressIndicator(
            modifier = Modifier.size(24.dp),
            color = MaterialTheme.colorScheme.onPrimary,
            strokeWidth = 2.dp
        )
    } else {
        Text("Registrarse")
```

```
    }
}
```

Feedback en Tiempo Real: La pantalla responde al estado isLoading del ViewModel. Cuando el usuario pulsa "Registrarse", el botón cambia su contenido por un CircularProgressIndicator y se deshabilitan todos los campos.
Justificación de UX: Esto evita que el usuario realice múltiples peticiones accidentales mientras se comunica con Firebase, garantizando una **interacción fluida y profesional.**

```
val msg: String? = authViewModel.errorMessage

// 2. Usamos un IF tradicional
if (msg != null) {
    Text(
        text = msg, // Al ser 'msg' una val local, el Smart Cast a
String es obligatorio
        color = MaterialTheme.colorScheme.error,
        modifier = Modifier.padding(top = 8.dp),
        style = MaterialTheme.typography.bodySmall
    )
}
```

Análisis Técnico: El código realiza un volcado del estado del ViewModel a una variable local. Esto permite a Kotlin realizar un Smart Cast seguro de String? a String, asegurando que la aplicación no sufra cierres inesperados al intentar renderizar un mensaje nulo. Los errores se presentan visualmente usando el color error de Material 3 para una identificación inmediata.

```
Button(
    onClick = {
        authViewModel.signUp(email, password, nombre) {
            // Si el registro tiene éxito, vamos al Mapa
            navController.navigate("mapa") {
                // Limpiamos el historial para que no pueda volver
atrás al registro
                popUpTo(Screens.Login.route) { inclusive = true }
            }
        }
    },
```

Nota sobre coordenadas: Tras tu reciente cambio en el NavGraph, aunque el código aquí aún pida el 0.0 de forma explícita, el sistema ya está preparado para priorizar los valores por defecto de Cádiz si se omite o se ajusta la ruta.
Gestión de la Pila (Backstack): El uso de popUpTo con inclusive = true es una medida de seguridad y usabilidad. Evita que un usuario recién registrado pueda volver atrás a la pantalla de registro pulsando el botón de retroceso, forzando una navegación lineal hacia la funcionalidad principal.

MapScreen.kt

```kotlin
@Composable
fun MapScreen(
    navController: NavHostController,
    mapViewModel: MapViewModel,
    authViewModel: AuthViewModel,
    latParam: Double,
    lngParam: Double
) {
    var selectedIncidencia by remember { mutableStateOf<Incidencia?>( value = null) }
    var showDetailDialog by remember { mutableStateOf( value = false) }
    // 1. Inicialización: Carga datos y detecta el Rol
    LaunchedEffect( key1 = Unit) {
        mapViewModel.cargarDatos()
    }

    // 2. Estado de la Cámara (Se centra si viene del AdminList)
    val cameraPositionState = rememberCameraPositionState {
        position = CameraPosition.fromLatLngZoom( target = LatLng( latitude = latParam,  longitude = lngParam),  zoom = if (latParam != 36.5271) 15f else 10f)
    }

    // Estados para el diálogo de nueva incidencia
    var showDialog by remember { mutableStateOf( value = false) }
    var tempLatLng by remember { mutableStateOf<LatLng?>( value = null) }
    var titulo by remember { mutableStateOf( value = "") }
    var descripcion by remember { mutableStateOf( value = "") }
    var showCreateDialog by remember { mutableStateOf( value = false) }
```

```kotlin
// Launcher para la Galería
val galleryLauncher = rememberLauncherForActivityResult(
    contract = ActivityResultContracts.GetContent()
) { uri: Uri? ->
    mapViewModel.imagenSeleccionadaUri = uri
}

Scaffold(
    floatingActionButton = {
        Column {
            // Solo el Admin ve el botón para ir a la lista de gestión
            if (mapViewModel.esAdmin) {
                SmallFloatingActionButton(
                    onClick = { navController.navigate( route = "admin_list") },
                    modifier = Modifier.padding(bottom = 8.dp)
                ) {
                    Icon( imageVector = Icons.Default.List, contentDescription = "Ver Lista")
                }
            }
        }
    }
) { innerPadding ->
    Box(Modifier
        .fillMaxSize()
        .padding( paddingValues = innerPadding) // <--- AQUÍ SE CORRIGE EL ERROR
    ) {
```

```
) { innerPadding ->
    Box(Modifier
        .fillMaxSize()
        .padding( paddingValues = innerPadding) // <--- AQUÍ SE CORRIGE EL ERROR
    ) {
        GoogleMap(
            modifier = Modifier.fillMaxSize(),
            cameraPositionState = cameraPositionState,
                onMapLongClick = { latLng ->
                tempLatLng = latLng
                showCreateDialog = true
            }
        ) {
            mapViewModel.listaIncidencias.forEach { incidencia ->
                Marker(
                    state = MarkerState(position = LatLng( latitude = incidencia.latitud,  longitude = incidencia.longitud)),
                    title = incidencia.titulo,
                    onClick = {
                        selectedIncidencia = incidencia
                        showDetailDialog = true
                        true
                    }
                )
            }
        }
```

```
        // Mantenemos el indicador de carga si existe
        if (mapViewModel.isUploading) {
            CircularProgressIndicator(Modifier.align(Alignment.Center))
        }
    }

    // --- EL DIÁLOGO PUEDE IR AQUÍ O FUERA DEL BOX ---
    if (showDetailDialog && selectedIncidencia != null) {
        AlertDialog(
            onDismissRequest = { showDetailDialog = false },
            title = { Text( text = selectedIncidencia?.titulo ?: "Detalle") },
            text = {
                Column(modifier = Modifier.fillMaxWidth()) {
                    if (!selectedIncidencia?.imagenUrl.isNullOrBlank()) {
                        AsyncImage(
                            model = selectedIncidencia?.imagenUrl,
                            contentDescription = "Imagen",
                            modifier = Modifier
                                .fillMaxWidth()
                                .height( height = 200.dp)
                                .clip( shape = RoundedCornerShape( size = 8.dp)),
                            contentScale = ContentScale.Crop
                        )
                        Spacer(Modifier.height( height = 12.dp))
                    }
```

```
                    }
                    Text( text = selectedIncidencia?.descripcion ?: "", style = MaterialTheme.typography.bodyMedium)
                    Spacer(Modifier.height( height = 8.dp))
                    Text( text = "Estado: ${selectedIncidencia?.estado?.uppercase()}", color = MaterialTheme.colorScheme.primary)
                }
            },
            confirmButton = {
                TextButton(onClick = { showDetailDialog = false }) { Text( text = "Cerrar") }
            }
        )
    }
}
if (showCreateDialog) {
    AlertDialog(
        onDismissRequest = { showCreateDialog = false },
        title = { Text( text = "Nueva Incidencia") },
        text = {
            Column {
                TextField(value = titulo, onValueChange = { titulo = it }, label = { Text( text = "Título") })
                Spacer(Modifier.height( height = 8.dp))
                TextField(value = descripcion, onValueChange = { descripcion = it }, label = { Text( text = "Descripción") })
                Spacer(Modifier.height( height = 16.dp))

                Button(onClick = { galleryLauncher.launch( input = "image/*") }) {
                    Text( text = if (mapViewModel.imagenSeleccionadaUri != null) "Foto lista" else "Añadir Foto")
                }
```


```
val cameraPositionState = rememberCameraPositionState {
   position = CameraPosition.fromLatLngZoom(LatLng(latParam,
lngParam), if (latParam != 36.5271) 15f else 10f)
}
```

Análisis de Parámetros: La pantalla recibe latParam y lngParam. Tras la reciente configuración, si el usuario abre el mapa por primera vez, la cámara se centra en **Cádiz (36.5271, -6.2886)** con un zoom general (10f). Si se navega desde la lista de administración hacia una incidencia concreta, el zoom aumenta (15f) para localizar el punto exacto.
Eficiencia: El estado de la cámara se "recuerda" (rememberCameraPositionState), lo que evita que el mapa se resetee si el usuario rota la pantalla o despliega un diálogo.

```
mapViewModel.listaIncidencias.forEach { incidencia ->
       Marker(
           state = MarkerState(position =
LatLng(incidencia.latitud, incidencia.longitud)),
           title = incidencia.titulo,
           onClick = {
               selectedIncidencia = incidencia
               showDetailDialog = true
               true
           }
       )
   }
}
```

Lógica Reactiva: Se itera sobre la lista de incidencias que el MapViewModel mantiene actualizada por el flujo (Flow) de Firestore.
Interacción Natural (NUI): Al pulsar un marcador, se dispara un evento que captura la incidencia seleccionada y despliega un AlertDialog con el detalle, permitiendo una navegación intuitiva sin salir del contexto del mapa.

```
GoogleMap(
   modifier = Modifier.fillMaxSize(),
   cameraPositionState = cameraPositionState,
       onMapLongClick = { latLng ->
       tempLatLng = latLng
       showCreateDialog = true
```

```
    }
```

La aplicación utiliza la pulsación larga para obtener coordenadas. Al capturar el objeto LatLng del mapa, se garantiza que la incidencia se guarde en la ubicación geográfica exacta donde el ciudadano detectó el problema.

```
AsyncImage(
    model = selectedIncidencia?.imagenUrl,
    contentDescription = "Imagen",
    modifier = Modifier
        .fillMaxWidth()
        .height(200.dp)
        .clip(RoundedCornerShape(8.dp)),
    contentScale = ContentScale.Crop
)
```

Carga Asíncrona: Se integra la librería **Coil** a través de AsyncImage. Esto permite que las imágenes alojadas en **Firebase Storage** se descarguen y muestren de forma eficiente, gestionando automáticamente la caché y evitando que la app se ralentice al cargar fotos de alta resolución.

```
floatingActionButton = {
    Column {
        // Solo el Admin ve el botón para ir a la lista de gestión
        if (mapViewModel.esAdmin) {
            SmallFloatingActionButton(
                onClick = { navController.navigate("admin_list") },
                modifier = Modifier.padding(bottom = 8.dp)
            ) {
                Icon(Icons.Default.List, contentDescription = "Ver
Lista")
            }
        }
    }
}
```

El FloatingActionButton (botón de lista) solo se renderiza si mapViewModel.esAdmin es verdadero. Esto asegura que un ciudadano común no pueda acceder a las herramientas de gestión, reforzando la seguridad de la interfaz desde el propio diseño declarativo.

AdminListScreen.kt

```kotlin
@OptIn( ...markerClass = ExperimentalMaterial3Api::class)
@Composable
fun AdminListScreen(
    navController: NavHostController,
    mapViewModel: MapViewModel
) {
    // 1. CARGA DE DATOS: Sincronizado con el nombre definitivo del ViewModel
    LaunchedEffect( key1 = Unit) {
        mapViewModel.cargarDatos()
    }

    val incidencias = mapViewModel.listaIncidencias

    Scaffold(
        topBar = {
            TopAppBar(
                title = { Text( text = "Gestión de Incidencias (Admin)") },
                colors = TopAppBarDefaults.topAppBarColors(
                    containerColor = MaterialTheme.colorScheme.primaryContainer
                )
            )
        }
    ) { padding ->
        if (incidencias.isEmpty()) {
            Box(Modifier.fillMaxSize().padding( paddingValues = padding), contentAlignment = Alignment.Center) {
                Text( text = "No hay incidencias registradas en el sistema")
            }
        } else {
```

```kotlin
            }
        } else {
            LazyColumn(
                modifier = Modifier.fillMaxSize().padding( paddingValues = padding),
                contentPadding = PaddingValues( all = 16.dp),
                verticalArrangement = Arrangement.spacedBy( space = 12.dp)
            ) {
                items( items = incidencias) { incidencia ->
                    IncidenciaItem(
                        incidencia = incidencia,
                        onApprove = {
                            // 2. ACCIÓN: Sincronizado con cambiarEstado del ViewModel
                            mapViewModel.cambiarEstado(incidencia,  nuevoEstado = "aprobada")
                        },
                        onViewOnMap = {
                            // 3. NAVEGACIÓN: Sincronizado con los argumentos del NavGraph
                            navController.navigate( route = "mapa?lat=${incidencia.latitud}&lng=${incidencia.longitud}")
                        }
                    )
                }
            }
        }
    }
```

```
@Composable
fun IncidenciaItem(
    incidencia: Incidencia,
    onApprove: () -> Unit,
    onViewOnMap: () -> Unit
) {
    Card(
        modifier = Modifier.fillMaxWidth(),
        elevation = CardDefaults.cardElevation( defaultElevation = 4.dp)
    ) {
        Column(modifier = Modifier.padding( all = 16.dp)) {
            Row(verticalAlignment = Alignment.CenterVertically) {
                Column(modifier = Modifier.weight( weight = 1f)) {
                    Text( text = incidencia.titulo, style = MaterialTheme.typography.titleMedium)
                    Text(
                        text = incidencia.estado.uppercase(),
                        color = if (incidencia.estado == "aprobada") Color( color = 0xFF4CAF50) else Color.Gray,
                        style = MaterialTheme.typography.labelSmall
                    )
                }
                // Botón para ir al mapa y ver dónde está el problema
                IconButton(onClick = onViewOnMap) {
                    Icon( imageVector = Icons.Default.LocationOn, contentDescription = "Ver en mapa", tint = Color.Red)
                }
            }

            // Imagen de la incidencia (si existe)
            if (!incidencia.imagenUrl.isNullOrBlank()) {
                Spacer(Modifier.height( height = 8.dp))
                AsyncImage(
                    model = incidencia.imagenUrl,
                    contentDescription = null,
```

```
                AsyncImage(
                    model = incidencia.imagenUrl,
                    contentDescription = null,
                    modifier = Modifier
                        .fillMaxWidth()
                        .height( height = 180.dp)
                        .clip( shape = RoundedCornerShape( size = 8.dp)),
                    contentScale = ContentScale.Crop
                )
            }

            Spacer(Modifier.height( height = 8.dp))
            Text( text = incidencia.descripcion, style = MaterialTheme.typography.bodyMedium)

            // Espaciador y botón de aprobación (solo si está pendiente)
            if (incidencia.estado != "aprobada") {
                Spacer(Modifier.height( height = 12.dp))
                Button(
                    onClick = onApprove,
                    modifier = Modifier.fillMaxWidth(),
                    colors = ButtonDefaults.buttonColors(containerColor = Color( color = 0xFF4CAF50))
                ) {
                    Icon( imageVector = Icons.Default.Check, contentDescription = null)
                    Spacer(Modifier.width( width = 8.dp))
                    Text( text = "Aprobar Incidencia")
                }
            }
        }
    }
```

```
LazyColumn(
    modifier = Modifier.fillMaxSize().padding(padding),
    contentPadding = PaddingValues(16.dp),
    verticalArrangement = Arrangement.spacedBy(12.dp)
```

```
) {
    items(incidencias) { incidencia ->
        IncidenciaItem(
            incidencia = incidencia,
            onApprove = {
                // 2. ACCIÓN: Sincronizado con cambiarEstado del
ViewModel
                mapViewModel.cambiarEstado(incidencia, "aprobada")
            },
            onViewOnMap = {
                // 3. NAVEGACIÓN: Sincronizado con los argumentos
del NavGraph

navController.navigate("mapa?lat=${incidencia.latitud}&lng=${incid
encia.longitud}")
            }
        )
    }
}
```

Análisis de Rendimiento : A diferencia de una columna estándar, LazyColumn solo renderiza los elementos que son visibles en pantalla. Esto es fundamental para la eficiencia, ya que permite que la aplicación gestione cientos de reportes en Cádiz sin consumir memoria excesiva ni degradar la experiencia de usuario.

Aprobación de Contenido: Al pulsar "Aprobar Incidencia", se invoca a mapViewModel.cambiarEstado(incidencia, "aprobada"). Gracias al uso de Flows en el repositorio, este cambio se refleja instantáneamente en el mapa de todos los usuarios.

Feedback Visual: El estado de la incidencia se muestra mediante etiquetas de color (Verde para aprobadas, Gris para pendientes), facilitando una interacción natural (NUI) y rápida para el administrador.

```
onViewOnMap = {
    // 3. NAVEGACIÓN: Sincronizado con los argumentos del NavGraph

navController.navigate("mapa?lat=${incidencia.latitud}&lng=${incid
encia.longitud}")
}
```

Sincronización de Coordenadas: Esta es una de las funciones más potentes de la interfaz. Al pulsar el icono de ubicación, la app redirige al gestor a la MapScreen, pasando la latitud y longitud exactas como argumentos.
Experiencia de Usuario: Como vimos en la documentación de MapScreen, la cámara detecta estos parámetros y hace un "zoom in" automático sobre la incidencia, permitiendo al administrador inspeccionar visualmente el entorno del reporte antes de validarlo.

```
fun IncidenciaItem(
```

```
    incidencia: Incidencia,
    onApprove: () -> Unit,
    onViewOnMap: () -> Unit
```

Arquitectura de UI: Se utiliza un componente separado para cada tarjeta de la lista. Esto sigue el principio de responsabilidad única, haciendo que el código sea más legible y mantenible.

```
if (!incidencia.imagenUrl.isNullOrBlank()) {
    Spacer(Modifier.height(8.dp))
    AsyncImage(
        model = incidencia.imagenUrl,
        contentDescription = null,
        modifier = Modifier
            .fillMaxWidth()
            .height(180.dp)
            .clip(RoundedCornerShape(8.dp)),
        contentScale = ContentScale.Crop
    )
}
```

Consumo de Multimedia: Integra AsyncImage para mostrar la evidencia fotográfica subida por el ciudadano. La visualización de la imagen es condicional (if (!incidencia.imagenUrl.isNullOrBlank())), lo que asegura que el diseño de la tarjeta no se rompa si un reporte no incluye foto.