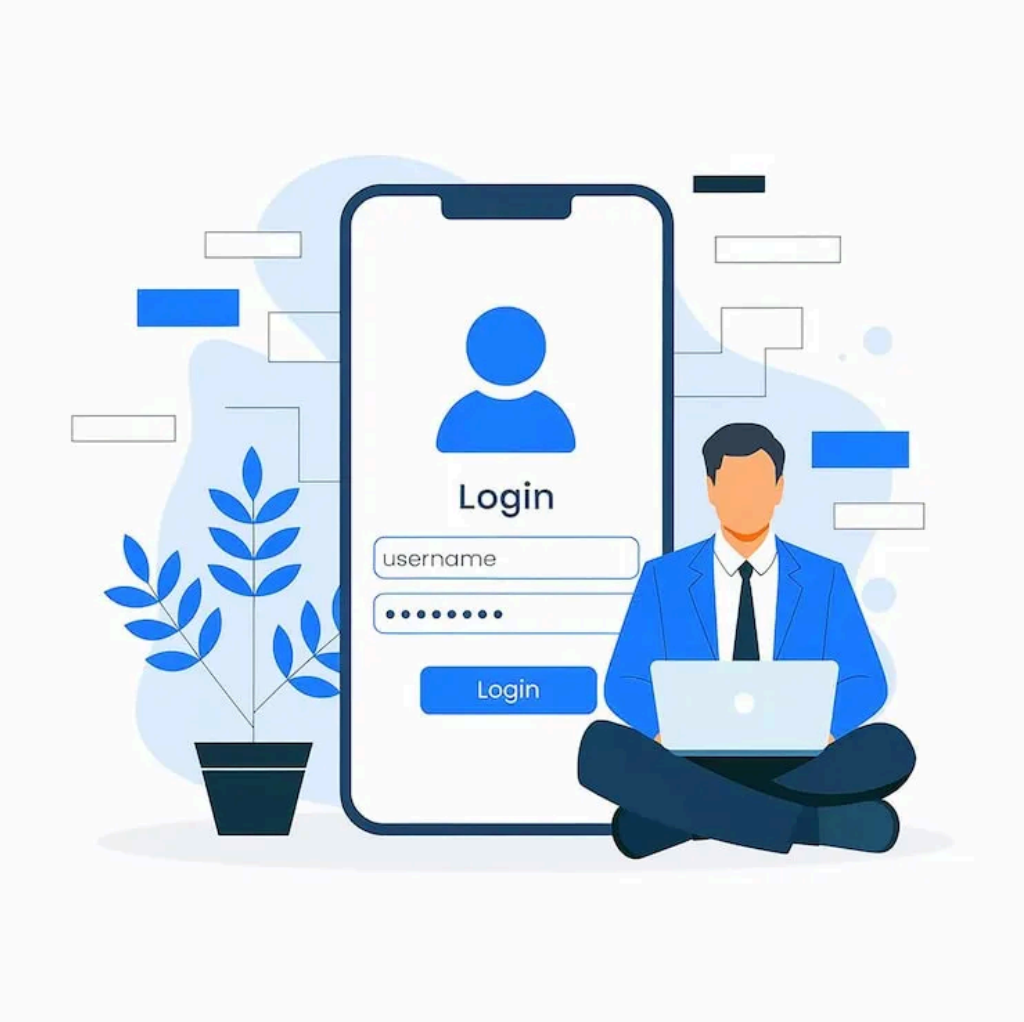
Investigación Teórica Login



Índice

1. ¿Qué es un login?

Definición

Proceso encargado de controlar el acceso a un sistema informático mediante un sistema de autentificación que implementa credenciales otorgadas por el usuario.

De esta manera, se puede controlar quién accede a la información o sistema y de qué manera lo hace, dependiendo del rol de usuario que tenga.

Dicho sistema posee unos campos de uso común para identificar al usuario que intenta autenticarse. Suelen ser:

- Nombre de usuario: El usuario debe escribir el nombre con el que le gustaría sentirse identificado y tratado.
- Correo electrónico: El usuario debe escribir el correo electrónico de su propiedad.
- Contraseña: El usuario debe escribir su clave para poder acceder al sistema de forma única.

Validaciones mínimas

Para este sistema de autenticación, existen varias validaciones que el usuario debe de cumplir para poder acceder al sistema:

- Campo obligatorio: El campo marcado con dicho atributo debe de ser rellenado.
- Formato válido: El campo rellenado debe de cumplir con un formato válido para poder evitar errores. Se suele relacionar dicho atributo al campo de correo electrónico, aunque hoy en día también se suele pedir en la contraseña con fines de seguridad.
- Longitud: Relacionado con el formato, el campo debe de tener una mínima cantidad de caracteres al ser rellenado.
- Manejo correcto de errores: Si el usuario rellena de forma incorrecta un campo, el sistema debe proporcionar suficiente información para que este pueda identificar el problema y arreglarlo.

State Hoisting

Patrón de diseño que consiste en pasar el estado como parámetro a los componentes en lugar de mantenerse dentro de ellos, es decir, elevar el estado.

Este patrón se necesita en los formularios ya que permite validar múltiples campos juntos, enviar valores al mismo tiempo para la autenticación, mostrar errores de forma coordinada, etc.

Gracias a esto, los formularios simplifican validaciones y envíos para poder hacer más eficiente el sistema de login.

Autenticación local

Este tipo de autenticación ocurre dentro del propio sistema o aplicación, ya que este valida con su propia base de datos las credenciales del usuario y compara usuario y contraseña de forma local.

Esta autenticación está presente en un login de una aplicación web con usuario y contraseña propios o cuando un usuario es guardado en una base de datos de la propia app.

Autenticación remota

Este otro tipo de autenticación delega la validación a un servicio externo, ya que el usuario se autentica en otro sistema que confirma la identidad del usuario.

Aplicaciones web grandes como Google, Microsoft o Facebook son ejemplos de autenticación remota.

Diferencias entre autenticaciones

Las principales diferencias son:

- Lugar de validación: La local lo valida en su propio sistema mientras que la remota lo hace a través de un servicio externo.
- Base de datos: Sucede lo mismo, la local posee una propia en su sistema pero la remota no, tiene una externa.
- Gestión: La local debe de ser gestionada por tí mismo. En cambio la remota no hace falta que la gestiones tú.
- Disponibilidad: Al ser local, es independiente, pero si es remota, dependerá del proveedor.

---

2. Perfiles de usuario y roles

Definición de rol

Un rol son permisos y responsabilidades que son asignados a un determinado usuario dentro de un sistema. Con este, el usuario puede ver ciertas cosas, hacer algunas y otras no, etc.

Un usuario suele tener solo un rol, aunque dependiendo del sistema puede tener varios.

Tipos de rol

Los roles más comunes son:

- Usuario normal: Acceso básico al sistema, sin funcionalidades exclusivas.
- Usuario administrador: Control total o bastante avanzado del sistema.
- Usuario editor: Este usuario puede modificar el contenido del sistema.
- Usuario moderador: Gestiona los roles y usuarios o contenido del sistema.

Ejemplos de apps que usan roles

Varias redes sociales, como Facebook o Instagram, emplean este tipo de autenticación, dividiendo los roles en usuario, moderador o administrador.

Algunas plataformas educativas, como Moodle o Google Classroom, usan roles como estudiante y docente que son prácticamente idénticos a los roles comunes, solo que enfocados en un contexto diferente.

Diferencias entre usuarios

Dependiendo del tipo de rol que tenga el usuario, podrá ver o gestionar dentro del sistema diferentes cosas:

- Acceso al sistema: Ambos usuarios pueden.
- Gestionar o manipular la información: Solo el usuario administrador, el usuario normal solo podrá verla.
- Gestión de usuarios: Sólo podrá el usuario administrador.

Permisos de los tipos de usuario

Acerca del proyecto de la Pokédex, el usuario normal sólo podrá ver la información de los Pokémon, buscar algún Pokémon manualmente, guardar en favoritos algunos, cambiar de sección, etc. Si este usuario intenta acceder a pantallas donde no tiene acceso, saldrá un error.

Sobre el usuario administrador, además de todos los permisos del usuario normal, podrá crear, editar o eliminar Pokémon a su gusto y acceder pantallas exclusivas como la de estadísticas.

3. Arquitectura para un login

Composables que usaría

A la hora de crear un login en Android Studio, es necesario recurrir a ciertos composables que nos ayudarán a hacerlo estético y agradable visualmente. Los composables serían:

- Text: Servirá para poder introducir texto.
- Column: Este composable organizará los elementos verticalmente.
- Spacer: Gracias a este composable entre los elementos habrá espacio para que no se vean demasiado juntos.
- TextField: Se usará con el fin de que el usuario introduzca nombre de usuario o email. Este composable también permite tener un borde de color, por lo que cuando sea incorrecta la credencial, podría tornarse de un color rojo.
- PasswordField: Similar a TextField, en este se introducirá la contraseña del usuario, ya que este composable oculta los caracteres por motivos de seguridad.
- Button: Este composable se activa cuando es pulsado, por lo que servirá como envío de datos para la autenticación.
- CircularProgressIndicator: Simulará una carga cuando el botón sea pulsado.

Estados necesarios

El login necesitará unas variables o estados que simulen ser las credenciales:

- Nombre de usuario: Guarda lo que el usuario escribe mediante ***var nombreUsuario by rememberSaveable { mutableStateOf("") }.***
- Contraseña: Guarda la clave del usuario mediante ***var contrasena by rememberSaveable { mutableStateOf("") }.***
- Error: Guarda el mensaje de error y lo muestra o no mediante ***var mensajeError by rememberSaveable { mutableStateOf<String?>(null) }.***
- Carga: Indica que el login está en proceso mediante ***var carga by rememberSavebale { mutableStateOf(false) }.***

Navegación según el rol

Una vez el usuario se haya autenticado en el login, accede al sistema como usuario normal o como usuario administrador.

Dependiendo del rol que tenga el usuario tendrá disponibles algunas opciones y pantallas ya comentadas anteriormente, es decir, manipular información de Pokémons o bien acceder a la pantalla de estadísticas.

Esto se manejaría gracias a un NavHost que dependiendo del rol que tenga el usuario le permitirá o no acceder a dichas funcionalidades.

Estructura de carpetas

Con el fin de lograr una administración correcta de contenido, este proyecto debería de seguir una estructura de carpetas basada en:

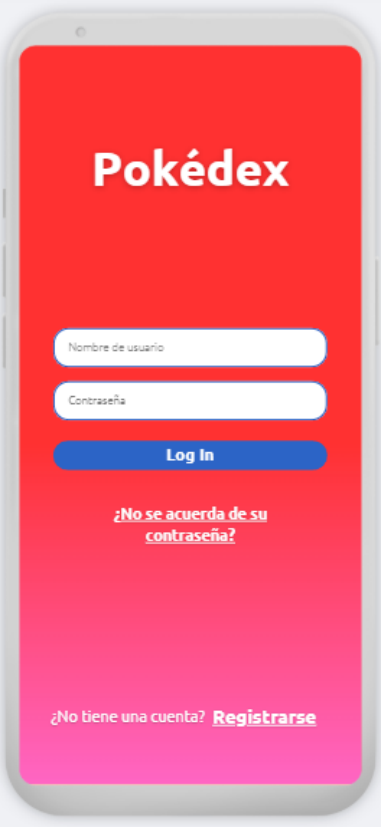
- Model: Clases que representan la información de la app.
- Repository: Elementos que se encargan de proporcionar datos.
- Ui:
    - Components: Componentes reutilizables que se usan en diferentes pantallas.
    - Screens: Pantallas que conforman la app.
    - Navigation: Elementos que definen cómo se mueve el usuario por la app.

Las pantallas de login y de la app estarían dentro de screens, mientras que el NavHost estaría dentro de navigation.

4. Diseño previo

Esquema visual de login

Para el login, he creado 4 diferentes pantallas:



Esta pantalla simula ser el propio sistema de login. En ella, el usuario debe introducir las credenciales que usó para registrarse en el sistema.

El usuario debe introducir su nombre de usuario y su contraseña.

Si estas no coinciden o no están registradas, saldrá un mensaje de error debajo de los respectivos campos, indicando el fallo.

Si el usuario es nuevo y no tiene ninguna cuenta, podrá registrarse e introducir sus credenciales por primera vez.

Al pulsar el botón de Log In, el usuario accede al sistema.

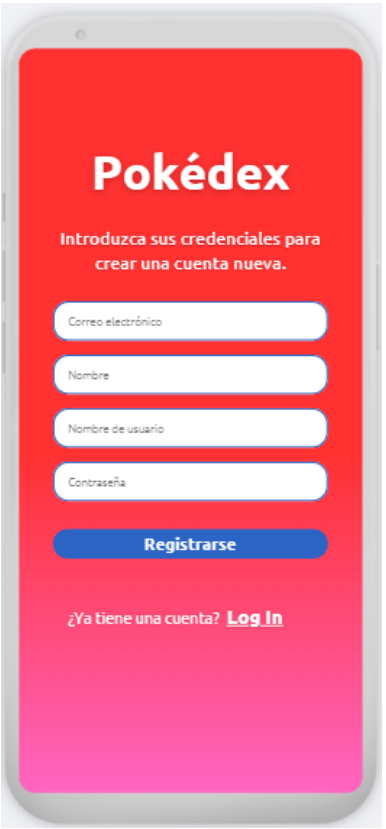
Si el usuario nunca se ha registrado en el sistema, debe introducir sus credenciales por primera vez.

Deberá escribir su correo electrónico, su nombre, su nombre de usuario y su contraseña.

Al igual que en la primera pantalla, si hay algún error en los campos, aparecerán diversos errores dependiendo del campo mal introducido.

Si el usuario ya tiene una cuenta, pulsará Log In para acceder a la pantalla principal.

Al pulsar el botón de registrarse, el usuario accede al sistema.



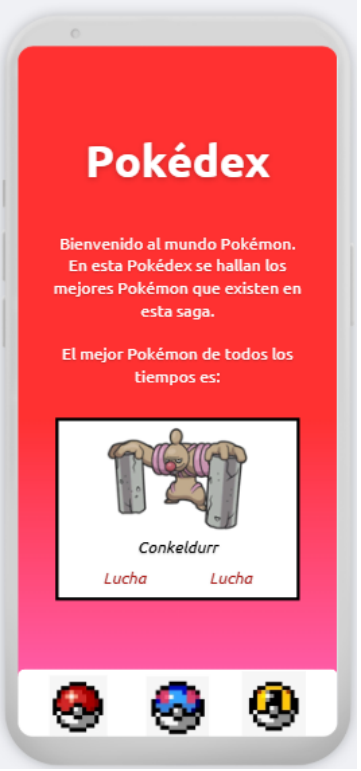
Pantallas para usuario normal

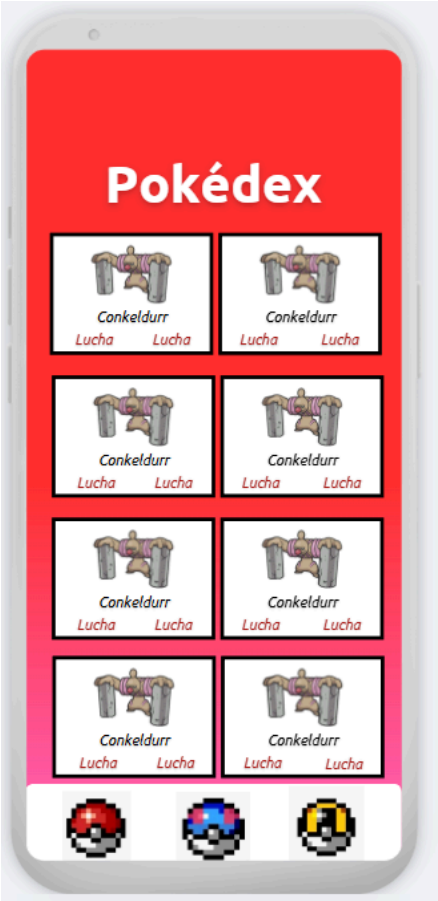
Cuando el usuario se autentica correctamente en el sistema a través del login, lo primero que ve es esta pantalla.

Esta pantalla muestra el mejor Pokémon de todos. Este, siendo usuario administrador, puede ser modificado a su antojo.

Como usuario normal, solo podemos apreciar la información y navegar a las demás pantallas a través de las Pokéballs que están abajo en la pantalla.

Según la Pokéball clickada, navegará a una pantalla determinada. La básica mostrará la pantalla presente. La Super Ball mostrará varios Pokémon organizados en rejilla. La Ultra Ball mostrará los Pokémon organizados por tipos.





Esta pantalla almacena y muestra todos los Pokémon registrados en la Pokédex en forma de rejilla.

Los Pokémon son mostrados según su especie y tipos.

Al pulsar en un Pokémon, se mostrará un pop up sobre más datos del Pokémon correspondiente, como su generación, peso o etapa evolutiva.

Mantiene los tres botones de navegación para visitar las diferentes pantallas de la Pokédex.

Esta tercera pantalla simula ser un filtro de Pokémon por tipo.

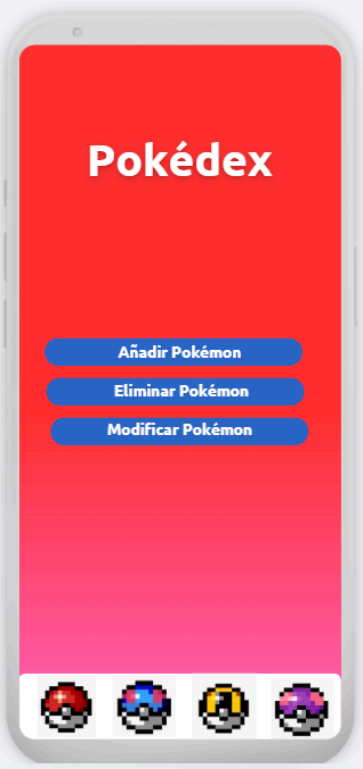
Según el tipo del Pokémon, aparecerá o no en el apartado.

Si el Pokémon posee dos tipos, aparecerá en ambos tipos.

Al igual que en la organización por rejilla, al pulsar sobre un Pokémon, se mostrarán sus datos.

Pantallas exclusivas para usuario administrador

El usuario administrador solo poseerá el permiso para acceder a una única pantalla exclusiva, en la que podrá manipular ciertos Pokémon y sus datos, además de añadir o eliminar Pokémon.



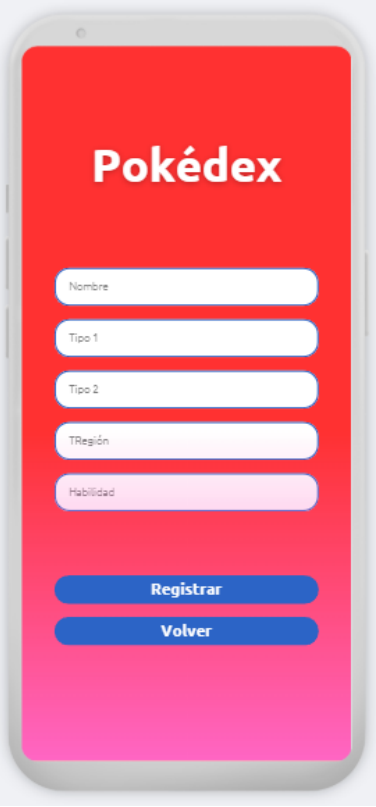
Siendo administrador, le aparecerá una Master Ball con la que podrá acceder a la pantalla exclusiva.

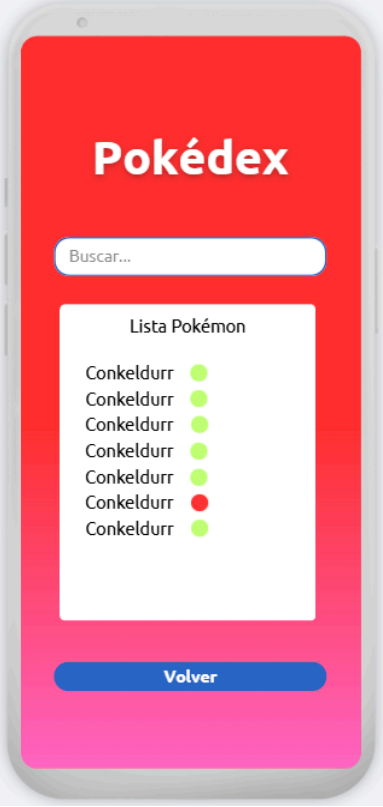
Se mostrarán tres botones diferentes.

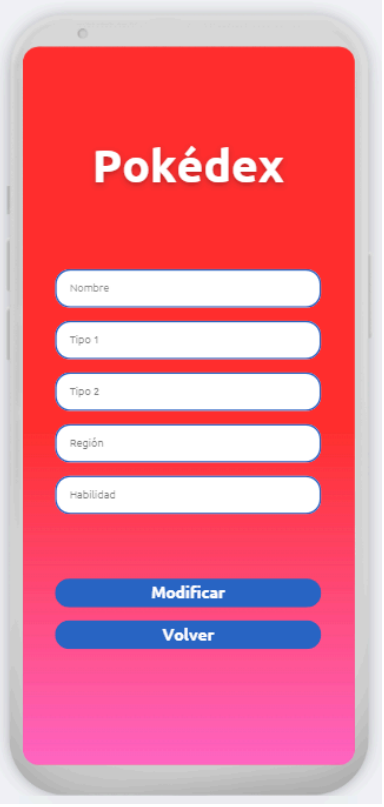
El botón de añadir Pokémon, como su nombre indica, te navegará a una ventana en la que deberás rellenar datos para crear uno nuevo.

El segundo botón navega a una pantalla con todos los Pokémon y podrás elegir cuál eliminar.

El tercero mostrará todos los Pokémon, donde se elige uno y se podrán modificar sus datos.







La primera pantalla ofrece varios campos a rellenar para añadir el Pokémon.

La segunda pantalla ofrece una lista con todos los Pokémon registrados. Si quieres eliminar uno, pulsa el botón verde que pasará a ser rojo.

La tercera pantalla ofrece la misma pantalla de registro de Pokémon, solo que los campos están ya rellenados con lo ya establecidos. Al cambiar uno, el administrador deberá pulsar el botón para guardar los cambios.

Bloqueo a pantallas restringidas

Según el tipo de usuario que se haya registrado, le aparecerá la Master Ball para poder editar información del sistema.

Esto será controlado por una variable que recogerá si el usuario, al iniciar sesión, es administrador o no.

De esta manera, si la variable es true, aparecerá la Master Ball. De lo contrario, no aparecerá la posibilidad de abrir la pantalla.

En el sistema solo habrá un usuario administrador registrado, que será inicializado al inicio del programa. Sus credenciales serán básicas y siempre las mismas. Si las credenciales introducidas coinciden con las inicializadas, la variable de esAdministrador será true.

---

5. Conclusiones

Dificultades de implementación

La organización de archivos debe ser clara y concisa, por lo que será difícil organizarlo de forma perfecta y sin confusiones.

Acerca del flujo del programa, lo más difícil para mi nivel, sería mandar un correo a una dirección introducida por el usuario. Es por esto que lo he cambiado a que solo envíe la dirección, y si hay una registrada, cambie la contraseña asignada a dicho correo. Hay errores de seguridad en dicha funcionalidad pero como es a nivel local y personal, no habrá ningún problema.

Sobre el diseño no creo que hayan muchos inconvenientes, ya que son elementos básicos.

Ventajas de uso

Para que la información esté protegida y no pueda ser manipulada por cualquier persona, es importante establecer roles en un sistema de login.

Estableciendo roles como nivel de usuario normal y otro de administrador, podemos corregir este conflicto de seguridad. Si el usuario es normal, no podrá acceder a dichas pantallas o manipular la información de la app. Si al contrario, es administrador, podrá modificar la información a su antojo.

---

6. Capturas de pantalla y explicaciones

Carpeta model

La data class Pokemon sirve para poder crear instancias de Pokémon a lo largo de la aplicación.

```
data class Pokemon(
    val nombre: String,
    val tipo1: Tipos,
    val tipo2: Tipos,
    val descripcion: String,
    val generacion: Generacion,
    @DrawableRes var foto: Int
)
```

```
enum class Tipos(val color: Color) {
    1 Usage
    NORMAL(Color( color = 0xFFA8A77A)),
    3 Usages
    FUEGO(Color( color = 0xFFEE8130)),
    2 Usages
    AGUA(Color( color = 0xFF6390F0)),
    2 Usages
    PLANTA(Color( color = 0xFF7AC74C)),
    ELECTRICO(Color( color = 0xFFF7D02C)),
    3 Usages
```

La enum class de Tipos almacena todos los tipos de Pokémon disponibles y guardan el color que se corresponde con cada tipo en la saga principal.

```
enum class Generacion {
    1 Usage
    KANTO,
    1 Usage
    JOHTO,
    2 Usages
    HOENN,
```

La enum class Generacion almacena todas las regiones de los juegos principales de la saga.

```
open class PantallaBase(val rutaActual: String)
```

La clase PantallaBase sirve para poder ser llamada para especificar la ruta actual del sistema, define las rutas del sistema.

```
class Pantalla {
    4 Usages
    object Principal: PantallaBase( rutaActual = "principal")
    3 Usages
    object Rejilla: PantallaBase( rutaActual = "rejilla")
    3 Usages
    object Filtrado: PantallaBase( rutaActual = "filtrado")
    2 Usages
    object Admin: PantallaBase( rutaActual = "admin")
}
```

La clase Pantalla es un contenedor de pantallas principales de la app. Cada objeto que tiene Pantalla representa una pantalla que hereda de PantallaBase usando la ruta a la que asociamos. Esto evita problemas de escritura y evita tener que escribir directamente en el NavHost la ruta.

```
sealed class PantallaAdminRutas(val ruta: String) {
    1 Usage
    object Menu: PantallaAdminRutas( ruta = "admin")
    2 Usages
    object Anadir: PantallaAdminRutas( ruta = "admin/anadir")
    2 Usages
    object Eliminar: PantallaAdminRutas( ruta = "admin/eliminar")
}
```

La clase sellada PantallaAdminRutas es un contenedor exclusivo de rutas del modo admin.

La clase sellada PantallasApp simula lo mismo que la anterior clase, solo que exclusivo para las pantallas de login, registro y el Home.

```
sealed class PantallasApp(val ruta: String) {
    4 Usages
    object Login: PantallasApp( ruta = "login")
    2 Usages
    object Registro: PantallasApp( ruta = "register")
    3 Usages
    object Home: PantallasApp( ruta = "home")
}
```

La data class SesionPersistida guarda el usuario actual, si hay una sesión iniciada, y los usuarios que se han registrado, llamando a ObjetoUsuario.Usuario. Todo esto lo guarda en formato JSON posteriormente.

```
data class SesionPersistida(
    val usuarioActual: String?,
    val usuarios: List<ObjetoUsuario.Usuario>
)
```

Por último, he creado un objeto llamado ObjetoUsuario que almacena todo lo relacionado con el Login o el registro.

```
data class Usuario(
    val email: String = "",
    val nombre: String = "",
    val nombreUsuario: String,
    val contrasena: String,
    val admin: Boolean = false
)
```

Primero, guarda la data class Usuario para poder crear instancias de este, con todos sus datos.

```
private val usuarioAdmin = Usuario(
    nombreUsuario = "usuarioAdmin",
    contrasena = "123456",
    admin = true
)
```

Luego, por defecto, guarda credenciales de administrador, para cuando el usuario ponga estas credenciales, pueda ser admin.

Aquí se guardan en la propia clase los usuario registrados y el usuario actual registrado / logueado.

En el bloque init se inicializa siempre el usuario admin para que pueda loguearse siempre.

```
private val usuariosRegistrados = mutableListOf<Usuario>()

6 Usages
var usuarioActual: Usuario? = null
    private set

init {
    usuariosRegistrados.add(usuarioAdmin)
}
```

Estas funciones se usarán más tarde en la Pantalla del Login o del Registro. Guarda todas las credenciales y las registra en la propia clase, retornando si se ha podido autenticar o no.

```
fun login(nombreUsuario: String, contrasena: String): Boolean {
    val usuario = usuariosRegistrados.find {
        it.nombreUsuario == nombreUsuario && it.contrasena == contrasena
    }
    usuarioActual = usuario
    return usuario != null
}

1 Usage
fun registro(usuario: Usuario): Boolean {
    if (usuariosRegistrados.any { it.nombreUsuario == usuario.nombreUsuario }) {
        return false
    }
    usuariosRegistrados.add(usuario)
    usuarioActual = usuario
    return true
}
```

Esta función verifica que el usuario es admin o no.

```
fun esAdmin(): Boolean = usuarioActual?.admin == true
```

Estas funciones guardan en un archivo JSON usando la clase SesionPersistida el usuario actual y los registrados en el sistema. Todo esto lo logra gracias al objeto SesionJsonManager, que realiza todo el guardado de datos.

```
fun guardarEstado(context: Context) {
    SesionJsonManager.guardar(
        context,
        SesionPersistida(
            usuarioActual = usuarioActual?.nombreUsuario,
            usuarios = usuariosRegistrados
        )
    )
}

fun cargarEstado(context: Context) {
    val sesion = SesionJsonManager.cargar(context) ?: return
    usuariosRegistrados.clear()
    usuariosRegistrados.addAll( elements = sesion.usuarios)

    usuarioActual = sesion.usuarioActual?.let { nombre ->
        usuariosRegistrados.find { it.nombreUsuario == nombre }
    }
}
```

Esta función sirve para cerrar sesión del usuario cambiándolo a null, también llamando a SesionJsonManager.

```
fun logout(context: Context) {
    usuarioActual = null
    SesionJsonManager.borrarSesion(context)
}
```

Carpeta repository

En este directorio solo se ubica la clase PokemonRepository, que guarda una sola función llamada getPokemon, la cual retorna una lista con todos los Pokémon registrados en la Pokédex, con sus respectivos datos. Esta función se ubica en el companion object porque es un método estático, es decir, no requiere ningún estado de instancia, porque no tendría sentido crear una instancia de la clase cada vez que quieras mostrar los Pokémon.

```
class PokemonRepository {
    4 Usages
    companion object{
        4 Usages
        fun getPokemon(): List<Pokemon>{
            return listOf(
                Pokemon(
                    nombre = "Conkeldurr",
                    tipo1 = Tipos.LUCHA,
                    tipo2 = Tipos.LUCHA,
                    descripcion = "Albañil musculoso con dos vigas de piedra",
                    Generacion.TESELIA,
                    foto = R.drawable.conkeldurr
                ),
                Pokemon(
```

Carpeta ui/components

Esta carpeta también guarda únicamente un solo archivo, llamado PokemonComponent, cuya única función, que se llama igual, muestra en un Card los datos del Pokémon. Además, cuando se pulsa sobre el Card, se muestra un AlertDialog que muestra aún más datos del Pokémon.

```
var mostrarDialog by rememberSaveable() { mutableStateOf( value = false) }
```

```
if (mostrarDialog) {
    AlertDialog(
        onDismissRequest = { mostrarDialog = false },
        confirmButton = {
            TextButton(onClick = { mostrarDialog = false }) {
                Text( text = "Cerrar")
            }
        },
        title = { Text(text = pokemon.nombre) },
        text = {
            Column(
```

Cuando se clica cerrar, sale del AlertDialog turnándose false.

Carpeta ui/data

Aquí solo se ubica la clase SesionJsonManager, que como su nombre indica, controla la sesión del usuario en archivos JSON que se declaran en la clase.

```
private const val FILE_NAME = "usuarios.json"
2 Usages
private val gson = Gson()
```

Posee unas funciones, como una privada que obtiene el archivo JSON y lo retorna.
Otra función guarda la sesión del usuario en el sistema usando el archivo Json y escribiendo en este.

```
private fun obtenerArchivoJson(context: Context): File {
    return File( parent = context.filesDir, child = FILE_NAME)
}

2 Usages
fun guardar(context: Context, sesion: SesionPersistida) {
    val json = gson.toJson( src = sesion)
    obtenerArchivoJson(context).writeText( text = json)
}
```

La función cargar carga los datos del archivo Json y dependiendo si existe retorna un objeto SesionPersistida con los datos de los usuarios registrado y el usuario autenticado en el momento.
La función borrarSesión vuelve el usuario actual como nulo para que haga log out del sistema.

```
fun cargar(context: Context): SesionPersistida? {
    val file = obtenerArchivoJson(context)
    if (!file.exists()) return null
    return gson.fromJson( json = file.readText(), classOfT = SesionPersistida::class.java)
}

1 Usage
fun borrarSesion(context: Context) {
    val sesion = cargar(context) ?: return
    guardar(
        context,
        sesion = sesion.copy(usuarioActual = null)
    )
}
```

Carpeta ui/navigation

Se ubican varios archivos en esta ruta.

El primero es FooterNavigation.

```
val pantallas = mutableListOf(
    Pantalla.Principal,
    Pantalla.Rejilla,
    Pantalla.Filtrado
)
```

Este crea una lista mutable con las pantallas básicas que estarán en la barra de navegación inferior.

Si el usuario es Administrador, añade a dicha lista una nueva pantalla, la de administrador.

```
if (ObjetoUsuario.esAdmin()) {
    pantallas.add(Pantalla.Admin)
}
```

Gracias a NavigationBar, podemos crear el menú de navegación en la barra inferior de la pantalla, ya que lo pasaremos más adelante como bottomBar.
Crea una variable actual que guarda la ruta de la pantalla que está en ese momento visible y presente.
Seguidamente, itera sobre la lista de pantallas creando las rutas según el tipo de pantalla que sea y usando iconos específicos.

```
NavigationBar {
    val actual = navController.currentBackStackEntryAsState().value?.destination?.route

    pantallas.forEach { pantalla ->
        val icono = when (pantalla) {
            Pantalla.Principal -> R.drawable.pokeball
            Pantalla.Rejilla -> R.drawable.superball
            Pantalla.Filtrado -> R.drawable.ultraball
            Pantalla.Admin -> R.drawable.masterball
            else -> R.drawable.pokeball
        }
```

Por último, crea objetos de navegación por cada objeto de la lista de pantallas.

```
NavigationBarItem(
    selected = actual == pantalla.rutaActual,
    onClick = {
        navController.navigate( route = pantalla.rutaActual
        ) },
    icon = {
        Image(
            painter = painterResource( id = icono),
            contentDescription = pantalla.rutaActual,
            modifier = Modifier.size( size = 28.dp)
        )
    }
)
```

El archivo PantallaAdmin posee dos rutas de navegación según el botón que sea pulsado. Uno navega hasta la pantalla de añadir Pokémon y el otro hacia la pantalla de eliminar Pokémon. Esto se logra llamando la ruta establecida en la clase PantallaAdminRutas.

```
TextButton(
    onClick = { navController.navigate( route = PantallaAdminRutas.Anadir.ruta) },
    colors = ButtonDefaults.buttonColors(
        containerColor = Color.Black,
        contentColor = Color.White
    )
) {
    Text( text = "Añadir Pokémon")
}
TextButton(
    onClick = { navController.navigate( route = PantallaAdminRutas.Eliminar.ruta) },
    colors = ButtonDefaults.buttonColors(
        containerColor = Color.Black,
        contentColor = Color.White
    )
) {
    Text( text = "Eliminar Pokémon")
}
```

El archivo PantallaNavegacion es un contenedor de navegación interno dentro de la sección Home de la app.

Recibe una variable onLogout que es llamada cuando el usuario cierra sesión en el sistema.

Usa como bottomBar la función de FooterNavegacion explicada anteriormente.

```
@Composable
fun PantallaNavegacion(
    onLogout: () -> Unit
) {
    val navController = rememberNavController()

    Scaffold(
        bottomBar = {
            FooterNavegacion(navController)
        }
```

Tiene su propio NavController para manejar la navegación entre las pantallas internas de la aplicación.

Cada composable representa una pantalla interna de Home que puede ser mostrada, incluyendo las visibles y las de administrador.

Este no gestiona el Log In ni el registro, ya que hay otro archivo que ya lo hace.

```
NavHost(
    navController = navController,
    startDestination = Pantalla.Principal.rutaActual,
    modifier = Modifier.padding(paddingValues)
) {

    composable( route = Pantalla.Principal.rutaActual) {
        PantallaPrincipal(
            onLogout = onLogout
        )
    }

    composable( route = Pantalla.Rejilla.rutaActual) {
        PantallaRejilla()
    }

    composable( route = Pantalla.Filtrado.rutaActual) {
        PantallaFiltrado()
    }
```

El último archivo, AppNavHost, orquesta la navegación principal del sistema.

Este bloque se ejecuta solo una vez, cargando el archivo JSON y redirigiendo al usuario al Home según si ya hay uno registrado o no.

```
@Composable
fun AppNavHost() {
    val navController = rememberNavController()
    val context = LocalContext.current

    LaunchedEffect( key1 = Unit) {
        ObjetoUsuario.cargarEstado(context)

        if (ObjetoUsuario.usuarioActual != null) {
            navController.navigate( route = PantallasApp.Home.ruta) {
                popUpTo( route = PantallasApp.Login.ruta) { inclusive = true }
            }
        }
    }
```

```
composable( route = PantallasApp.Login.ruta) {
    PantallaLogin(
        loginCorrecto = {
            navController.navigate( route = PantallasApp.Home.ruta) {
                popUpTo( route = PantallasApp.Login.ruta) { inclusive = true }
            }
        },
        botonRegistro = {
            navController.navigate( route = PantallasApp.Registro.ruta)
        }
    )
}

composable( route = PantallasApp.Registro.ruta) {
    PantallaRegistro(
        registroCorrecto = {
            navController.navigate( route = PantallasApp.Home.ruta) {
                popUpTo( route = PantallasApp.Login.ruta) { inclusive = true }
            }
        },
        loginCambio = {
            navController.popBackStack()
        }
    )
}
```

Estos compostables dentro del NavHost validan si el login o el registro son correctos, saltando hacia el Home si es así.

Para terminar, este último composable llama a PantallaNavegacion para la otra navegación. Si el usuario cierra sesión, llama al ObjetoUsuario y navega a la pantalla de log in.

```
composable( route = PantallasApp.Home.ruta) {
    val context = LocalContext.current
    PantallaNavegacion(
        onLogout = {
            ObjetoUsuario.logout(context)
            navController.navigate( route = PantallasApp.Login.ruta) {
                popUpTo( route = PantallasApp.Home.ruta) { inclusive = true }
            }
        }
    )
}
```

Carpeta ui/screens

Aquí se guardan todas las pantallas posibles de mi aplicación.

La primera pantalla en aparecer es PantallaLogin que recibe dos corrutinas (loginCorrecto y botonRegistro) que son llamadas en la navegación principal de AppNavHost. Guarda variables sin información que después recibirán los valores de los OutlinedTextField.
Si el login es correcto mediante las variables de usuario y contraseña, guarda en JSON las credenciales que recibe y llama a la corrutina.

```
@Composable
fun PantallaLogin(
    loginCorrecto: () -> Unit,
    botonRegistro: () -> Unit
) {
    var usuario by rememberSaveable { mutableStateOf( value = "") }
    var contrasena by rememberSaveable { mutableStateOf( value = "") }
    var error by rememberSaveable { mutableStateOf( value = "") }

    val contexto = LocalContext.current
    if (ObjetoUsuario.login( nombreUsuario = usuario, contrasena)) {
        ObjetoUsuario.guardarEstado( context = contexto)
        loginCorrecto()
    }
```

Aquí se actualizan las variables del inicio con su nuevo valor.

```
OutlinedTextField(
    value = usuario,
    onValueChange = { usuario = it },
```

El botón de login mira si se puede hacer el login. Si está bien, llama la corrutina, sino lanza un error.

```
Button(
    onClick = {
        if (ObjetoUsuario.login( nombreUsuario = usuario, contrasena)) {
            loginCorrecto()
        } else {
            error = "Usuario o contraseña incorrectos"
        }
    },
```

La otra corrutina es llamada en el botón de registrarse por si no tiene cuenta.

```
TextButton(
    onClick = botonRegistro
) {
    Text(
        text = "¿No tiene una cuenta? Registrarse",
        color = Color.Black
    )
}
```

La PantallaRegistro sigue la misma lógica, guarda variables y se le pasan dos corrutinas.

```
@Composable
fun PantallaRegistro(
    registroCorrecto: () -> Unit,
    loginCambio: () -> Unit
) {
    var email by rememberSaveable { mutableStateOf( value = "") }
    var nombre by rememberSaveable { mutableStateOf( value = "") }
    var nombreUsuario by rememberSaveable { mutableStateOf( value = "") }
    var contrasena by rememberSaveable { mutableStateOf( value = "") }
    var error by rememberSaveable { mutableStateOf( value = "") }
```

Aquí se ven las validaciones de los OutlinedTextField.

```
Button(
    onClick = {
        when {
            email.isBlank() -> {
                error = "El correo electrónico no puede estar vacío"
            }
            nombre.isBlank() -> {
                error = "El nombre no puede estar vacío"
            }
            nombreUsuario.isBlank() -> {
                error = "El nombre de usuario no puede estar vacío"
            }
            contrasena.isBlank() -> {
                error = "La contraseña no puede estar vacía"
            }
            contrasena.length < 6 -> {
                error = "La contraseña debe tener al menos 6 caracteres"
            }
```

```
else -> {
    val success = ObjetoUsuario.registro(
        usuario = ObjetoUsuario.Usuario(
            email = email,
            nombre = nombre,
            nombreUsuario = nombreUsuario,
            contrasena = contrasena,
            admin = false
        )
    )
    if (success) {
        ObjetoUsuario.guardarEstado(context)
        registroCorrecto()
    } else {
        error = "El usuario ya existe"
    }
}
```

Si no hay errores de validación, crea una variable llamada success de tipo boolean que intenta registrar usando el ObjetoUsuario al nuevo usuario con las credenciales que ha introducido. Nunca será administrador, por eso es false.

Si es true, lo guarda en el archivo JSON. Sino, muestra un error.

```
TextButton(
    onClick = {
        ObjetoUsuario.logout(context)
        onLogout()
    }
) {
    Text(
        text = "Cerrar sesión",
        color = Color.White,
        fontWeight = FontWeight.Bold
    )
}
```

La PantallaPrincipal muestra un título, dos textos, un botón arriba a la derecha y una llamada a PokemonComponente.

Este TextButton permite recrear el cierre de sesión si se pulsa en él, llamando a ObjetoUsuario y a la corrutina pasada en el constructor de la función.

He decidido que el Pokémon que aparece sea aleatorio, mediante esta variable, donde se usa un .random() sobre la lista de Pokémon.

```
val pokemonAleatorio = rememberSaveable {
    PokemonRepository.getPokemon().random()
}
```

Luego, se muestra el componente usando dicha variable.

```
PokemonComponent(pokemon = pokemonAleatorio)
```

La PantallaRejilla muestra todos los Pokémon de la lista

```
val listaPokemon = PokemonRepository.getPokemon()
```

de dos a dos gracias a GridCells.Fixed(2), usando LazyVerticalGrid, que organiza en rejilla a los Pokémon.

Consigue mostrar a todos iterando sobre la propia lista y mostrando cada Pokémon usando el componente.

```
LazyVerticalGrid(
    state = lazyGridState,
columns = GridCells.Fixed( count = 2)
) {
    items( items = listaPokemon) { pokemon ->
        PokemonComponent(pokemon)
    }
}
```

La última pantalla del sistema de usuario normal es la PantallaFiltrado, que muestra los Pokémon filtrando por tipos.

Primero, crea instancias de la lista de los Pokémon del repositorio y se llama una única vez, en una nueva variable llamada pokemonPorTipo.

```
val pokemons = PokemonRepository.getPokemon()
 💡
3 Usages
@OptIn( ...markerClass = ExperimentalFoundationApi::class)
@Composable
fun PantallaFiltrado() {
    val pokemonPorTipo = remember {
        agruparPokemonPorTipoCompleto(pokemons)
    }
```

Esta variable llama a una función privada del propio archivo que recibe una lista de Pokémon y devuelve un mapa que tiene como clave el tipo y como valor una lista de Pokémon que tienen ese tipo.

En la primera línea se crea un mapa donde cada tipo apunta a una lista vacía de Pokémon.

Luego, se itera sobre la lista filtrada por tipos y se crea una variable que es un set y combina si los dos tipos son duplicados. Para cada tipo en ese set añade el Pokémon a la lista correspondiente en el mapa.

```
private fun agruparPokemonPorTipoCompleto(listaPokemon: List<Pokemon>): Map<Tipos, List<Pokemon>> {
    val mapa = Tipos.entries.associateWith { mutableListOf<Pokemon>() }

    listaPokemon.forEach { pokemon ->
        val tiposUnicos = setOf(pokemon.tipo1, pokemon.tipo2)
        tiposUnicos.forEach { tipo ->
            mapa[tipo]?.add(pokemon)
        }
    }
    return mapa
}
```

En la función principal del archivo se itera sobre la lista de pokemonPorTipo y crea un stickyHeader por cada tipo. Usa la variable de la enum class Tipos, la del color, para darle el fondo a este header usando un Box y dándole como color de fondo el respectivo del tipo.

```
pokemonPorTipo.forEach { (tipo, listaPokemon) ->
    stickyHeader {
        Box(
            modifier = Modifier
                .fillMaxWidth()
                .background( color = tipo.color)
                .padding( all = 12.dp),
            contentAlignment = Alignment.Center
        ) {
            Text(
                text = tipo.name,
                color = Color.White,
                fontWeight = FontWeight.Bold,
                fontSize = 20.sp
            )
        }
    }
```

Si no hay Pokémon de un tipo, como el eléctrico, muestra un mensaje de eso mismo, no hay Pokémon de X tipo.

```
if (listaPokemon.isEmpty()) {
    item {
        Box(
            modifier = Modifier
                .fillMaxWidth()
                .padding( all = 24.dp),
            contentAlignment = Alignment.Center
        ) {
            Text(
                text = "No hay Pokémon de tipo ${tipo.name}",
                color = Color.White,
                fontSize = 16.sp,
                textAlign = TextAlign.Center
            )
        }
    }
```

Las últimas dos pantallas son del modo de administrador, tanto añadir como eliminar Pokémon.

La PantallaAnadirPokemon guarda variables de los campos a rellenar para crear el Pokémon. El resto de la función trata de los OutlinedTextField por cada campo, donde se actualizan las variables del inicio, y un botón no funcional porque lo he visto innecesario en cuanto a lo que se está evaluando en esta tarea.

```
@Composable
fun PantallaAnadirPokemon(){
    var nombre by rememberSaveable { mutableStateOf( value = "") }
    var tipo1 by rememberSaveable { mutableStateOf( value = "") }
    var tipo2 by rememberSaveable { mutableStateOf( value = "") }
    var region by rememberSaveable { mutableStateOf( value = "") }
    var descripcion by rememberSaveable { mutableStateOf( value = "") }
```

```
Button(
    onClick = { /* No hace nada */ },
    colors = ButtonDefaults.buttonColors(
        containerColor = Color.Black,
        contentColor = Color.White
    )
){
    Text(
        text = "Añadir Pokémon"
    )
}
```

La PantallaEliminarPokemon funciona de manera similar a PantallaRejilla. Muestra todos los Pokémon con sus datos, pero sin usar componentes ni en rejilla.

Los almacena todos en un Box de fondo blanco.

Crea una variable llamada pokemonVisibles que es un array que por cada uno almacena todos los datos del Pokémon gracias a *.

```
@Composable
fun PantallaEliminarPokemon() {
    val pokemonVisibles = rememberSaveable {
        mutableStateListOf(*PokemonRepository.getPokemon().toTypedArray())
    }
```

En la Box se muestra una imagen del Pokémon, su nombre, sus dos tipos en columna y un círculo verde. Este círculo sirve como botón para simular que se ha borrado un Pokémon, ya que al ser pulsado no se muestra en la pantalla.

He decidido usar otro tipo de lista en lugar de la original por motivos de simulación de eliminar Pokémon.

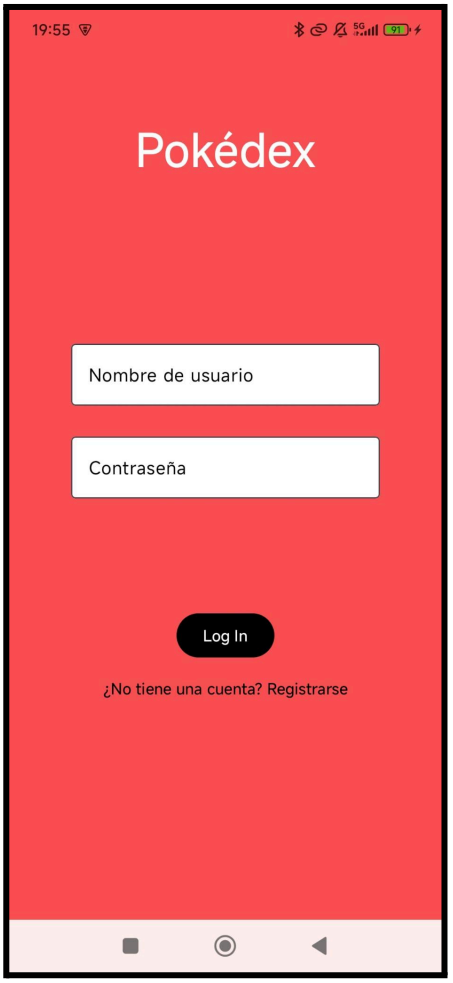
```
Box(
    modifier = Modifier
        .size( size = 24.dp)
        .background(Color.Green, CircleShape)
        .clickable {
            pokemonVisibles.remove( element = pokemon)
        }
)
```

Main Activity

Esta es la clase que se llama en el xml, por lo que será la que el sistema corre inicialmente. En esta he puesto a AppNavHost.

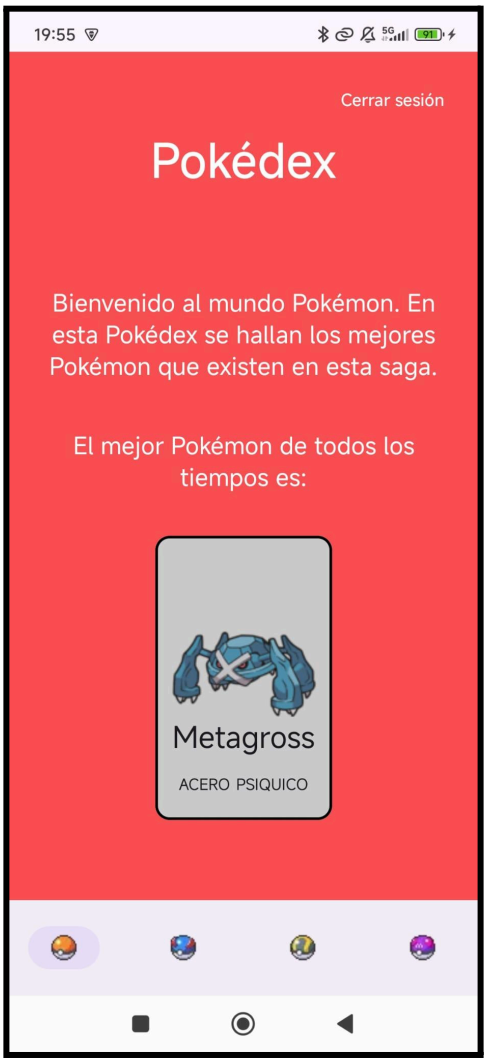
```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        enableEdgeToEdge()
        setContent {
            AppNavHost()
        }
    }
}
```

Pantalla Login



Pantalla Registro

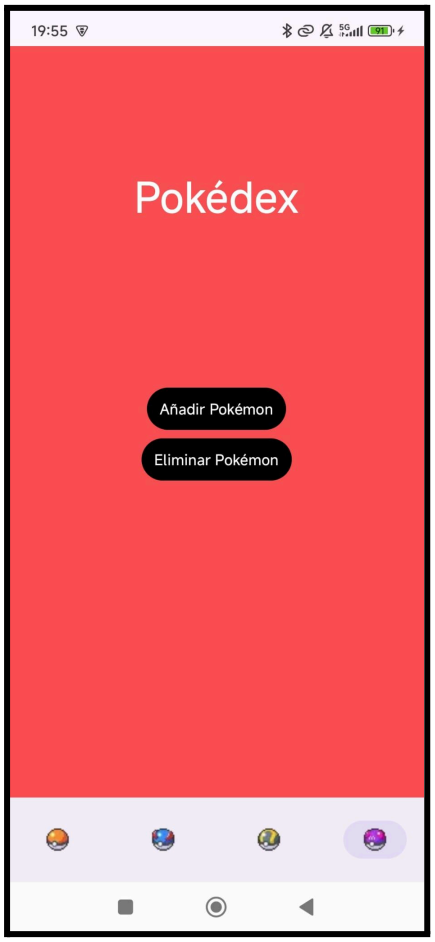
Pantalla Principal



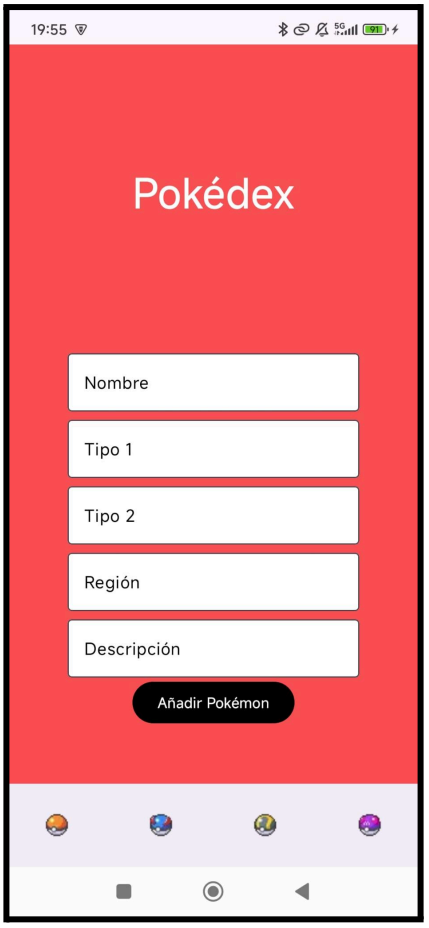
Pantalla Rejilla

Pantalla Filtrado

Pantalla Admin



Pantalla Añadir



Pantalla Eliminar