

Cambios Realizados para Login y CRUD de Usuarios

Resumen General

Este documento describe todos los cambios realizados para que el sistema de login funcione correctamente y las operaciones CRUD de usuarios se integren adecuadamente con el backend de Spring Boot.

Problemas Identificados y Solucionados

1. Login Fallaba con Tokens Nulos

Problema: El login retornaba `null` para tokens válidos porque el backend respondía con credenciales inválidas pero código 200.

Solución: Modificación en `LoginResponse.kt` para permitir campos opcionales:

```
@Serializable
data class LoginResponse(
    val token: String? = null,           // Permitir null
    val email: String? = null,           // Permitir null
    val message: String? = null,         // Permitir null
    val user: UserModel? = null         // Permitir null
)
```

Validación en `LevelUpRepository.kt`:

```
if (result is ApiResult.Success) {
    // Verificar que el token no sea nulo (credenciales válidas)
    if (result.data.token != null && result.data.email != null) {
        authManager.saveToken(result.data.token)
        authManager.saveUserEmail(result.data.email)
        return result
    } else {
        // El backend respondió 200 pero con credenciales inválidas
        return ApiResult.Error(Exception(result.data.message))
    }
}
```

2. Error "role is required" en Creación de Usuarios

Problema: El backend requería el campo `role` pero el frontend no lo enviaba correctamente.

Solución: Actualización de `RegisterRequest` en `ApiResponse.kt`:

```
@Serializable
data class RegisterRequest(
    val run: String,
    val firstName: String,
    val lastName: String,
    val email: String,
    val password: String,
    val role: String = "usuario" // Valor por defecto
)
```

Validación en UserViewModelWithApi.kt:

```
// Crear request directamente del UserModel actualizado
val request = RegisterRequest(
    run = usuario.run.trim(),
    firstName = usuario.firstName.trim(),
    lastName = usuario.lastName.trim(),
    email = usuario.email.trim(),
    password = usuario.password,
    role = usuario.role.isBlank { "usuario" } // Asignar rol por defecto
)
```

3. Incompatibilidad de Tipos ID (Int vs Long)

Problema: El frontend usaba **Int** para IDs pero el backend de Spring Boot usa **Long**.

Solución: Cambio en **User.kt**:

```
@Serializable
data class UserModel(
    val id: Long = 0L, // Cambio: Int -> Long
    val run: String = "",
    val direccion: String = "",
    val firstName: String = "",
    val lastName: String = "",
    val email: String = "",
    val password: String = "",
    val role: String = ""
) {
    // Computed properties para compatibilidad con UI existente
    val nombres: String get() = firstName
    val apellidos: String get() = lastName
    val correo: String get() = email
}
```

Actualización en ApiService.kt:

```

@PUT("api/v1/users/{id}")
suspend fun updateUser(
    @Header("Authorization") token: String,
    @Path("id") id: Long, // Cambio: Int -> Long
    @Body user: UserModel
): Response<UserUpdateResponse>

@DELETE("api/v1/users/{id}")
suspend fun deleteUser(
    @Header("Authorization") token: String,
    @Path("id") id: Long // Cambio: Int -> Long
): Response<UserDeleteResponse>

```

4. Alineación de Estructuras de Respuesta Backend-Frontend

Problema: El backend retorna respuestas estructuradas pero el frontend esperaba modelos directos.

Solución: Creación de clases de respuesta en [ApiResponse.kt](#):

```

@Serializable
data class UserCreateResponse(
    val message: String,
    val user: UserModel? = null,
    val error: String? = null
)

@Serializable
data class UserUpdateResponse(
    val message: String,
    val user: UserModel? = null,
    val error: String? = null
)

@Serializable
data class UserDeleteResponse(
    val message: String,
    val error: String? = null
)

```

Actualización del [ApiService.kt](#):

```

@POST("api/v1/users")
suspend fun createUser(
    @Header("Authorization") token: String,
    @Body user: RegisterRequest
): Response<UserCreateResponse> // Cambio: UserModel -> UserCreateResponse

@PUT("api/v1/users/{id}")

```

```
suspend fun updateUser(
    @Header("Authorization") token: String,
    @Path("id") id: Long,
    @Body user: UserModel
): Response<UserUpdateResponse> // Cambio: UserModel -> UserUpdateResponse
```

5. Corrección de Nombres de Campos

Problema: Inconsistencia entre nombres de campos del frontend y backend.

Solución: Actualización en `UserViewModel.kt`:

```
fun editarUsuario(id: Long, newRun: String, newDireccion: String,
                  newName: String, newApellido: String, newCorreo: String,
                  newPassword: String){
    viewModelScope.launch {
        val usuariosActuales = usuarios.value.toMutableList()
        val index = usuariosActuales.indexOfFirst { it.id == id }
        if (index != -1) {
            val usuarioEditado = usuariosActuales[index].copy(
                run = newRun,
                direccion = newDireccion,
                firstName = newName,           // Cambio: nombres -> firstName
                lastName = newApellido,         // Cambio: apellidos -> lastName
                email = newCorreo,             // Cambio: correo -> email
                password = newPassword
            )
            usuariosActuales[index] = usuarioEditado
            dataStorageManager.saveUsers(usuariosActuales)
        }
    }
}
```

6. Actualización del Repositorio para Nuevas Respuestas

Problema: El repositorio esperaba tipos de retorno incorrectos.

Solución: Actualización en `LevelUpRepository.kt`:

```
suspend fun createUser(user: RegisterRequest):
ApiResult<UserCreateResponse> {
    val token = getAuthToken() ?: return ApiResult.Error(Exception("No
authentication token"))

    val result = apiCall {
        apiService.createUser(authManager.getBearerToken(token), user)
    }

    if (result is ApiResult.Success) {
```

```

        // Refresh local data
        getUsers().collect {} // This will update local storage
    }

    return result
}

suspend fun updateUser(id: Long, user: UserModel): ApiResult<UserUpdateResponse> {
    // Similar pattern...
}

suspend fun deleteUser(id: Long): ApiResult<UserDeleteResponse> {
    // Similar pattern...
}

```

7. Manejo de Respuestas en ViewModels

Problema: Los ViewModels no manejaban las nuevas respuestas estructuradas.

Solución: Actualización en [UserViewModelWithApi.kt](#):

```

when (val result = repository.createUser(request)) {
    is ApiResult.Success -> {
        _isLoading.value = false
        playSuccessSound()
        println("DEBUG: Usuario creado exitosamente - ${result.data.message}")
        // Recargar la lista de usuarios desde el backend
        loadUsers()
    }
    is ApiResult.Error -> {
        _isLoading.value = false
        _errorMessage.value = result.exception.message
        println("DEBUG: Error al crear usuario - ${result.exception.message}")
    }
    is ApiResult.Loading -> {}
}

```

Estructura Final de Archivos

Archivos Modificados:

1. **ApiResponse.kt** - Nuevas clases de respuesta y RegisterRequest actualizado
2. **User.kt** - ID cambiado a Long, campos alineados con backend
3. **ApiService.kt** - Endpoints actualizados con tipos correctos
4. **LevelUpRepository.kt** - Métodos actualizados para nuevas respuestas
5. **UserViewModelWithApi.kt** - Manejo de respuestas estructuradas

6. UserViewModel.kt - Campos actualizados en editarUsuario

Endpoints del Backend Utilizados:

POST /api/auth/login	- Login de usuarios
POST /api/auth/register	- Registro desde pantalla pública
GET /api/v1/users	- Obtener lista de usuarios
POST /api/v1/users	- Crear nuevo usuario (desde admin)
PUT /api/v1/users/{id}	- Actualizar usuario existente
DELETE /api/v1/users/{id}	- Eliminar usuario

Flujo de Datos Actualizado

1. Login:

- Usuario ingresa credenciales → **LoginRequest**
- Backend valida → **LoginResponse** con token o mensaje de error
- Frontend verifica token no nulo antes de guardar

2. CRUD Usuarios:

- Crear: **RegisterRequest** → **UserCreateResponse**
- Leer: Token → **List<UserModel>**
- Actualizar: **UserModel** → **UserUpdateResponse**
- Eliminar: **id: Long** → **UserDeleteResponse**

Beneficios Obtenidos

- ✓ Login funciona con validación correcta de credenciales
- ✓ Creación de usuarios incluye rol requerido
- ✓ Tipos de ID consistentes (Long) entre frontend y backend
- ✓ Manejo adecuado de respuestas del backend
- ✓ Mensajes de éxito/error desde el backend
- ✓ Compatibilidad completa con Spring Boot JPA
- ✓ Eliminación de errores de compilación

Notas Importantes

- El campo **direccion** se mantiene solo en el frontend (no se envía al backend)
- Las computed properties (**nombres**, **apellidos**, **correo**) mantienen compatibilidad con UI existente
- Todos los IDs ahora usan **Long** para consistencia con JPA
- Las respuestas del backend incluyen mensajes descriptivos para mejor UX