ChargIST - EV Charging Station Management App

Final Project Report

Mobile and Ubiquitous Computing 24/25

Group 08
Afonso Pires 102803
Guilherme Leitão 99951
Diogo Marques 102760

Instituto Superior Técnico

June 2025

Contents

1 Data Schema and Server Endpoints

1.1 Backend Architecture

The application uses Firebase as its backend infrastructure, specifically leveraging Firestore for data persistence and Firebase Authentication for user management. This choice eliminates the need for custom server implementation while providing simplified real-time synchronization capabilities.

1.2 Data Schema

The Firestore database is structured with the following collections and documents:

1.2.1 Main Collections

1. **chargers** - Root collection for charging stations

```kotlin
// In Models.kt
data class Charger(
    val id: String,
    val name: String,
    val latitude: Double,
    val longitude: Double,
    val imageData: String?, // Base64 encoded
    val favoriteUsers: List<String>,
    val createdBy: String,
    val createdAt: Long,
    val updatedAt: Long?
)

```

2. **users** - User profiles and authentication data

```kotlin
// In User.kt
data class User(
    val id: String,
    val username: String,
    val createdAt: Long
)

```

1.2.2 Subcollections

Each charger document contains the following subcollections:

1. **chargingSlots** - Individual charging positions

```kotlin
// In Models.kt
data class ChargingSlot(
    val id: String,
    val chargerId: String,
    val speed: ChargingSpeed,
    val connectorType: ConnectorType,
    val available: Boolean,
    val damaged: Boolean,
    val price: Double,
    val updatedAt: Long
)

```

2. **ratings** - User ratings for the charging station
3. **paymentSystems** - Accepted payment methods

1.3 Repository Implementation

The data access is abstracted through repository interfaces, with Firebase implementations found in:

- `FireStoreRepository.kt` - Handles all charger-related operations
- `AuthRepository.kt` - Manages authentication flows

Key operations include, for example:

```
// In FireStoreRepository.kt
suspend fun createCharger(
    name: String,
    location: LatLng,
    imageData: String?,
    userId: String,
    chargingSlots: List<ChargingSlot>,
    paymentSystems: List<PaymentSystem>
): NetworkResult<Charger>
```

2 Android Maps API Integration

2.1 Maps Implementation

The application uses Google Maps Compose library (version 2.15.0) for map functionality, integrated primarily in:

- `HomeScreen.kt` - Main map view with charger markers
- `AddChargerScreen.kt` - Location selection for new chargers
- `ChargerDetailScreen.kt` - Static map preview

2.2 Key Features Implemented

2.2.1 Dynamic Marker Management

```
// In HomeScreen.kt
mapState.chargers.forEach { charger ->
                    val fav = userState.user?.id?.let { charger.
    favoriteUsers.contains(it) } == true
                    Marker(
                        state = MarkerState(charger.getLatLng()),
                        title = charger.name,
                        icon = BitmapDescriptorFactory.defaultMarker(
```

```kotlin
                            if (fav) BitmapDescriptorFactory.HUE_ROSE
                            else BitmapDescriptorFactory.HUE_GREEN
                        ),
                        onClick = {
                            onChargerClick(charger.id)
                            true
                        }
                    )
                }
}
```

2.2.2 Location Services Integration

The app uses FusedLocationProviderClient for accurate location tracking:

- Current location display with permission handling
- Automatic map centering on user location
- Location-based charger loading

2.2.3 Places API Integration - @GET("maps/api/place/nearbysearch/json")

Address search functionality implemented using Google Places API:

```kotlin
// In MapViewModel.kt
fun getAutocompleteSuggestions(query: String) {
    // ...
    val request = FindAutocompletePredictionsRequest.builder()
        .setQuery(query)
        .setCountries("PT")
        .setTypesFilter(listOf(PlaceTypes.ADDRESS))
        .build()
    // Process predictions...
}
```

3 Caching

3.1 Caching Implementation

3.1.1 Firestore Offline Persistence

Configured in `di/Modules.kt`:

```kotlin
// In Modules.kt
single {
    FirebaseFirestore.getInstance().apply {
        val settings = FirebaseFirestoreSettings.Builder()
            .setPersistenceEnabled(true)   // allow for local cache
            .build()
        firestoreSettings = settings
    }
}
```

This enables automatic caching of Firestore documents, allowing users to view previously loaded charging stations and their details offline, ensuring accessibility even in areas with poor connectivity.

3.1.2 Image Caching with Coil

Our App uses *Coil* library for efficient image caching:

```kotlin
// In ChargerDetailScreen.kt
AsyncImage(
    model = ImageRequest.Builder(LocalContext.current)
        .data(ImageCodec.base64ToBytes(photoData))
        .crossfade(true)
        .build(),
    contentDescription = "Charger image",
    // ...
)
```

3.1.3 Room Database Structure

Local database entities are defined with Room annotations for potential offline storage:

- `@Entity` annotations on data classes
- Foreign key relationships defined

4 Resource Frugality

4.1 Network Optimization

4.1.1 Image Compression

Before uploading, images are compressed and resized in `ImageStorageRepository.kt`:

```kotlin
suspend fun uriToBase64(context: Context, uri: Uri): String =
    withContext(Dispatchers.IO) {
        val bitmap = BitmapFactory.decodeStream(
            context.contentResolver.openInputStream(uri)
        )
        // Scale to max 1000px width
        val target = 1_000
        val scaled = bitmap.scale(
            target,
            (bitmap.height * target.toFloat() / bitmap.width).toInt()
        )
        // ...
        // Compress with 80% quality
        scaled.compress(Bitmap.CompressFormat.JPEG, 80, out)
        // ...
    }
```

4.1.2 Search Query Debouncing

To prevent excessive Places API calls, search queries are debounced:

```kotlin
// In AddChargerScreen.kt
LaunchedEffect(searchQuery) {
    delay(300) // Wait 300ms after last change
    if (text.isNotEmpty()) {
        mapViewModel.getAutocompleteSuggestions(searchQuery)
    }
}
```

4.1.3 Efficient Nearby Places Fetching

The app integrates with the Google Places API to retrieve nearby services such as restaurants, gas stations, stores, and cafes within a 500-meter radius of the selected charging station. This targeted fetching mechanism ensures that only relevant data is requested, minimizing API usage and reducing data transfer. Furthermore, distances to these places are calculated locally, eliminating the need for additional network requests.

4.1.4 Location-Based Charger Loading

To optimize data retrieval, the app loads only those charging stations located within a 500 meter radius of the current map center point. This approach significantly reduces the number of Firestore reads and enhances performance, particularly in regions with a high density of charging stations.

```kotlin
// In MapViewModel.kt
fun loadChargersNearLocation(center: LatLng, radiusMeters: Double = 500.0) {
        viewModelScope.launch {
            try {
                val bounds = createBoundsFromCenter(center, radiusMeters)
                chargerRepository.getChargersInBounds(bounds).collectLatest { chargers ->
                    _mapState.value = _mapState.value.copy(chargers = chargers, isLoading = false)
                }
            } catch (e: Exception) {
                _mapState.value = _mapState.value.copy(
                    error = "Error loading chargers: ${e.message}",
                    isLoading = false
                )
            }
        }
    }
```

4.2 Location Services Management

Location services are only activated after explicit permission:

```kotlin
// In HomeScreen.kt
val permissionLauncher = rememberLauncherForActivityResult(
    ActivityResultContracts.RequestPermission()
) { granted ->
    if (granted) mapViewModel.onLocationPermissionGranted()
}
```

4.3 UI Optimization

4.3.1 Lazy Loading

Lists use LazyColumn for efficient rendering:

```kotlin
LazyColumn(Modifier.fillMaxSize()) {
    items(searchState.searchResults) { result ->
        // Render only visible items
    }
}
```

4.3.2 State Management

Compose's state management prevents unnecessary recompositions:

- `remember` for local state
- `collectAsState()` for Flow observations

5 Advanced Features Implementation

5.1 User Accounts

Located in: `UserProfileScreen.kt`, `AuthRepository.kt`

5.1.1 Features

- Email/password authentication
- User profile creation with unique usernames
- Seamless login/logout flow
- Favorites synchronization across devices

5.1.2 Implementation Details

```kotlin
// In FirebaseAuthRepository class
override suspend fun signUp(
    username: String,
    email: String,
    pass: String
): NetworkResult<User> {
    auth.createUserWithEmailAndPassword(email, pass).await()
    val uid = auth.currentUser!!.uid
    val user = User(id = uid, username = username)
    users.document(uid).set(user).await()
    return NetworkResult.Success(user)
}
```

5.2 User Ratings

Located in: `ChargerDetailScreen.kt` (UI), `FirestoreRepository.kt` (backend)

5.2.1 Features

- 5-star rating system
- Rating histogram display
- Average rating calculation
- User can update their existing rating

5.2.2 Rating Statistics Implementation

```kotlin
// In ChargerDetailScreen.kt
@Composable
private fun RatingSection(
    ratingStats: RatingStats,
    userRating: Rating?,
    onRatingSubmit: (Int) -> Unit
) {
    // Display average rating
    Text(
        text = String.format("%.1f", ratingStats.averageRating),
        style = MaterialTheme.typography.headlineLarge
    )

    // Interactive star rating
    StarRating(
        rating = selectedRating.toFloat(),
        onRatingChanged = { newRating ->
            selectedRating = newRating.toInt()
        }
    )

    // Rating histogram
    RatingHistogram(
        ratingStats.histogram,
        ratingStats.totalRatings
    )
}
```

5.3 Social Sharing

- Share button in `ChargerDetailScreen.kt`
- Formats charging station information for sharing
- Uses Android's native share intent

5.4 Additional Infrastructure

We opted to expand the Nearby Services requirement by implementing it using the Google Places API and the following considerations:

- `NearbyPlacesRepository.kt` with Google Places API integration
- Fetches restaurants, gas stations, stores, and cafes within 500m
- Distance calculation using manual formula

6 Additional Notes

6.1 Architecture

MVVM Pattern - Clean separation of concerns:

- **Model**: Data classes and repositories
- **View**: Compose UI screens
- **ViewModel**: Business logic and state management

6.2 Security Considerations

API keys are hardcoded for demo purposes - we should use environment variables in a production scenario.

6.3 Future Enhancements

1. Implement remaining advanced features (Meta Moderation, Recommendations)
2. Add comprehensive error handling and retry mechanisms
3. Implement better offline App flow