Mobile-Specific Patterns & WebSocket Integration Deep Dive

1. Mobile-Specific Architecture Patterns

1.1 React Native Project Structure

```
mobile/
├── src/
│   ├── components/          # Shared UI components
│   │   ├── atoms/
│   │   │   ├── Button/
│   │   │   │   ├── Button.tsx
│   │   │   │   ├── Button.ios.tsx      # iOS-specific styles
│   │   │   │   ├── Button.android.tsx  # Android-specific styles
│   │   │   │   └── Button.test.tsx
│   │   │   └── Input/
│   │   ├── molecules/
│   │   └── organisms/
│   ├── screens/             # Screen components
│   │   ├── HomeScreen/
│   │   ├── EventScreen/
│   │   └── ProfileScreen/
│   ├── navigation/          # Navigation configuration
│   │   ├── AppNavigator.tsx
│   │   ├── AuthNavigator.tsx
│   │   └── types.ts
│   ├── services/            # API and business logic
│   │   ├── api/
│   │   ├── location/
│   │   ├── notifications/
│   │   └── storage/
│   ├── hooks/               # Custom hooks
│   │   ├── useAppState.ts
│   │   ├── useNetworkStatus.ts
│   │   └── usePermissions.ts
│   ├── native/              # Native module interfaces
│   │   ├── LocationModule/
│   │   ├── CameraModule/
│   │   └── BiometricModule/
│   ├── utils/               # Utilities
│   │   ├── platform.ts
│   │   ├── permissions.ts
│   │   └── storage.ts
│   └── types/               # TypeScript definitions
├── ios/                     # iOS native code
├── android/                 # Android native code
└── assets/                  # Static assets
```

1.2 Platform-Specific Component Pattern

typescript

```typescript
// components/atoms/Button/Button.tsx
import React from 'react';
import { Platform } from 'react-native';
import ButtonIOS from './Button.ios';
import ButtonAndroid from './Button.android';

interface ButtonProps {
  title: string;
  onPress: () => void;
  variant?: 'primary' | 'secondary';
  disabled?: boolean;
  loading?: boolean;
}

export const Button: React.FC<ButtonProps> = (props) => {
  // Platform-specific component rendering
  if (Platform.OS === 'ios') {
    return <ButtonIOS {...props} />;
  }

  return <ButtonAndroid {...props} />;
};

// Alternative approach using Platform.select
export const Button2: React.FC<ButtonProps> = (props) => {
  const ButtonComponent = Platform.select({
    ios: ButtonIOS,
    android: ButtonAndroid,
    default: ButtonAndroid, // fallback
  });

  return <ButtonComponent {...props} />;
};
```

1.3 Responsive Design Pattern

typescript

```tsx
// hooks/useResponsiveDimensions.ts
import { useState, useEffect } from 'react';
import { Dimensions, ScaledSize } from 'react-native';

interface ResponsiveDimensions {
  width: number;
  height: number;
  isTablet: boolean;
  isLandscape: boolean;
  scale: number;
}

export const useResponsiveDimensions = (): ResponsiveDimensions => {
  const [dimensions, setDimensions] = useState(() => {
    const { width, height, scale } = Dimensions.get('window');
    return {
      width,
      height,
      isTablet: Math.min(width, height) >= 768,
      isLandscape: width > height,
      scale
    };
  });

  useEffect(() => {
    const subscription = Dimensions.addEventListener('change', ({ window }) => {
      setDimensions({
        width: window.width,
        height: window.height,
        isTablet: Math.min(window.width, window.height) >= 768,
        isLandscape: window.width > window.height,
        scale: window.scale
      });
    });

    return () => subscription?.remove();
  }, []);

  return dimensions;
};

// Usage in components
const EventCard: React.FC<EventCardProps> = ({ event }) => {
```

```
  const { isTablet, isLandscape } = useResponsiveDimensions();

  const cardStyle = {
    width: isTablet ? (isLandscape ? '45%' : '48%') : '100%',
    marginBottom: isTablet ? 16 : 12,
  };

  return (
    <View style={[styles.card, cardStyle]}>
      {/* Card content */}
    </View>
  );
};
```

1.4 Navigation Patterns

typescript

```tsx
// navigation/AppNavigator.tsx
import React from 'react';
import { NavigationContainer } from '@react-navigation/native';
import { createNativeStackNavigator } from '@react-navigation/native-stack';
import { createBottomTabNavigator } from '@react-navigation/bottom-tabs';
import { createDrawerNavigator } from '@react-navigation/drawer';
import { Platform } from 'react-native';

import { useUserStore } from '@/stores/userStore';
import { useResponsiveDimensions } from '@/hooks/useResponsiveDimensions';

const Stack = createNativeStackNavigator();
const Tab = createBottomTabNavigator();
const Drawer = createDrawerNavigator();

export const AppNavigator: React.FC = () => {
  const { isAuthenticated } = useUserStore();
  const { isTablet } = useResponsiveDimensions();

  return (
    <NavigationContainer>
      {isAuthenticated ? (
        isTablet ? <TabletNavigator /> : <PhoneNavigator />
      ) : (
        <AuthNavigator />
      )}
    </NavigationContainer>
  );
};

// Phone navigation - Tab-based
const PhoneNavigator: React.FC = () => (
  <Tab.Navigator
    screenOptions={({ route }) => ({
      tabBarIcon: ({ focused, color, size }) => {
        return getTabIcon(route.name, focused, color, size);
      },
      tabBarActiveTintColor: '#007AFF',
      tabBarInactiveTintColor: 'gray',
      headerShown: false,
    })}
  >
    <Tab.Screen name="Home" component={HomeScreen} />
```

```
    <Tab.Screen name="Events" component={EventsScreen} />
    <Tab.Screen name="Map" component={MapScreen} />
    <Tab.Screen name="Profile" component={ProfileScreen} />
  </Tab.Navigator>
);

// Tablet navigation - Drawer-based
const TabletNavigator: React.FC = () => (
  <Drawer.Navigator
    screenOptions={{
      drawerType: 'permanent',
      drawerStyle: {
        width: 250,
      },
    }}
  >
    <Drawer.Screen name="Home" component={HomeScreen} />
    <Drawer.Screen name="Events" component={EventsScreen} />
    <Drawer.Screen name="Map" component={MapScreen} />
    <Drawer.Screen name="Profile" component={ProfileScreen} />
  </Drawer.Navigator>
);

// Deep linking configuration
const linking = {
  prefixes: ['myapp://', 'https://myapp.com'],
  config: {
    screens: {
      Home: 'home',
      Events: 'events',
      EventDetail: 'events/:eventId',
      Profile: 'profile',
    },
  },
};
```

1.5 Native Module Integration

typescript

```ts
// native/LocationModule/LocationModule.ts
import { NativeModules, Platform, PermissionsAndroid } from 'react-native';
import Geolocation from '@react-native-community/geolocation';

interface LocationCoordinates {
  latitude: number;
  longitude: number;
  accuracy: number;
  altitude?: number;
  heading?: number;
  speed?: number;
}

interface LocationOptions {
  enableHighAccuracy?: boolean;
  timeout?: number;
  maximumAge?: number;
}

class LocationService {
  private watchId: number | null = null;

  async requestPermission(): Promise<boolean> {
    if (Platform.OS === 'android') {
      try {
        const granted = await PermissionsAndroid.request(
          PermissionsAndroid.PERMISSIONS.ACCESS_FINE_LOCATION,
          {
            title: 'Location Permission',
            message: 'This app needs access to location to show nearby events.',
            buttonNeutral: 'Ask Me Later',
            buttonNegative: 'Cancel',
            buttonPositive: 'OK',
          }
        );
        return granted === PermissionsAndroid.RESULTS.GRANTED;
      } catch (err) {
        console.warn(err);
        return false;
      }
    }
    return true; // iOS handles permissions automatically
  }
```

```
getCurrentLocation(options?: LocationOptions): Promise<LocationCoordinates> {
  return new Promise((resolve, reject) => {
    Geolocation.getCurrentPosition(
      (position) => {
        resolve({
          latitude: position.coords.latitude,
          longitude: position.coords.longitude,
          accuracy: position.coords.accuracy,
          altitude: position.coords.altitude || undefined,
          heading: position.coords.heading || undefined,
          speed: position.coords.speed || undefined,
        });
      },
      (error) => reject(error),
      {
        enableHighAccuracy: options?.enableHighAccuracy ?? true,
        timeout: options?.timeout ?? 15000,
        maximumAge: options?.maximumAge ?? 10000,
      }
    );
  });
}

startWatchingLocation(
  onLocationUpdate: (location: LocationCoordinates) => void,
  onError: (error: any) => void,
  options?: LocationOptions
): void {
  this.watchId = Geolocation.watchPosition(
    (position) => {
      onLocationUpdate({
        latitude: position.coords.latitude,
        longitude: position.coords.longitude,
        accuracy: position.coords.accuracy,
        altitude: position.coords.altitude || undefined,
        heading: position.coords.heading || undefined,
        speed: position.coords.speed || undefined,
      });
    },
    onError,
    {
      enableHighAccuracy: options?.enableHighAccuracy ?? true,
      timeout: options?.timeout ?? 15000,
```

```
      maximumAge: options?.maximumAge ?? 10000,
      distanceFilter: 10, // Update every 10 meters
    }
   );
  }

  stopWatchingLocation(): void {
   if (this.watchId !== null) {
    Geolocation.clearWatch(this.watchId);
    this.watchId = null;
   }
  }
}

export const locationService = new LocationService();

// Hook for using location
export const useLocation = () => {
  const [location, setLocation] = useState<LocationCoordinates | null>(null);
  const [loading, setLoading] = useState(false);
  const [error, setError] = useState<string | null>(null);

  const getCurrentLocation = async () => {
   setLoading(true);
   setError(null);

   try {
    const hasPermission = await locationService.requestPermission();
    if (!hasPermission) {
     throw new Error('Location permission denied');
    }

    const currentLocation = await locationService.getCurrentLocation();
    setLocation(currentLocation);
   } catch (err) {
    setError(err instanceof Error ? err.message : 'Failed to get location');
   } finally {
    setLoading(false);
   }
  };

  return {
   location,
   loading,
```

```
    error,
    getCurrentLocation,
  };
};
```

1.6 Push Notifications Pattern

typescript

```typescript
// services/notifications/PushNotificationService.ts
import messaging, { FirebaseMessagingTypes } from '@react-native-firebase/messaging';
import { Platform, Alert, Linking } from 'react-native';
import AsyncStorage from '@react-native-async-storage/async-storage';

class PushNotificationService {
  private token: string | null = null;

  async initialize(): Promise<void> {
    // Request permission
    const authStatus = await messaging().requestPermission();
    const enabled =
      authStatus === messaging.AuthorizationStatus.AUTHORIZED ||
      authStatus === messaging.AuthorizationStatus.PROVISIONAL;

    if (!enabled) {
      Alert.alert(
        'Notifications Disabled',
        'Please enable notifications in settings to receive updates.',
        [
          { text: 'Cancel', style: 'cancel' },
          { text: 'Settings', onPress: () => Linking.openSettings() },
        ]
      );
      return;
    }

    // Get FCM token
    this.token = await messaging().getToken();
    await this.saveTokenToStorage(this.token);

    // Listen for token refresh
    messaging().onTokenRefresh(async (token) => {
      this.token = token;
      await this.saveTokenToStorage(token);
      await this.updateTokenOnServer(token);
    });

    // Handle foreground messages
    messaging().onMessage(this.handleForegroundMessage);

    // Handle background messages
    messaging().setBackgroundMessageHandler(this.handleBackgroundMessage);
```

```
    // Handle notification open app
    messaging().onNotificationOpenedApp(this.handleNotificationOpenedApp);

    // Check if app was opened from a notification
    const initialNotification = await messaging().getInitialNotification();
    if (initialNotification) {
      this.handleNotificationOpenedApp(initialNotification);
    }
  }

  private async saveTokenToStorage(token: string): Promise<void> {
    await AsyncStorage.setItem('fcm_token', token);
  }

  private async updateTokenOnServer(token: string): Promise<void> {
    try {
      // Send token to your backend
      await apiClient.post('/users/push-token', { token });
    } catch (error) {
      console.error('Failed to update push token on server:', error);
    }
  }

  private handleForegroundMessage = (message: FirebaseMessagingTypes.RemoteMessage) => {
    // Show in-app notification
    Alert.alert(
      message.notification?.title || 'Notification',
      message.notification?.body || 'You have a new notification',
      [
        { text: 'Dismiss', style: 'cancel' },
        {
          text: 'View',
          onPress: () => this.handleNotificationAction(message.data),
        },
      ]
    );
  };

  private handleBackgroundMessage = async (message: FirebaseMessagingTypes.RemoteMessage) => {
    console.log('Background message:', message);
    // Handle background processing if needed
  };
```

```
  private handleNotificationOpenedApp = (message: FirebaseMessagingTypes.RemoteMessage) => {
    this.handleNotificationAction(message.data);
  };

  private handleNotificationAction(data: any) {
    // Navigate based on notification data
    if (data?.type === 'event_update') {
      // Navigate to event screen
      navigationRef.navigate('EventDetail', { eventId: data.eventId });
    } else if (data?.type === 'new_message') {
      // Navigate to messages screen
      navigationRef.navigate('Messages', { conversationId: data.conversationId });
    }
  }

  async subscribeToTopic(topic: string): Promise<void> {
    await messaging().subscribeToTopic(topic);
  }

  async unsubscribeFromTopic(topic: string): Promise<void> {
    await messaging().unsubscribeFromTopic(topic);
  }

  getToken(): string | null {
    return this.token;
  }
}

export const pushNotificationService = new PushNotificationService();
```

2. WebSocket Integration Deep Dive

2.1 Mobile WebSocket Client

typescript

```typescript
// services/websocket/MobileSocketClient.ts
import { io, Socket } from 'socket.io-client';
import { AppState, AppStateStatus, NetInfo } from 'react-native';
import AsyncStorage from '@react-native-async-storage/async-storage';
import { useUserStore } from '@/stores/userStore';

interface SocketOptions {
  autoConnect?: boolean;
  reconnection?: boolean;
  reconnectionAttempts?: number;
  reconnectionDelay?: number;
  timeout?: number;
}

class MobileSocketClient {
  private socket: Socket | null = null;
  private connectionState: 'disconnected' | 'connecting' | 'connected' = 'disconnected';
  private appState: AppStateStatus = 'active';
  private isNetworkConnected = true;
  private messageQueue: Array<{ event: string; data: any }> = [];
  private heartbeatInterval: NodeJS.Timeout | null = null;
  private listeners: Map<string, Function[]> = new Map();

  constructor(private options: SocketOptions = {}) {
    this.setupAppStateHandling();
    this.setupNetworkHandling();
  }

  private setupAppStateHandling(): void {
    AppState.addEventListener('change', this.handleAppStateChange);
  }

  private setupNetworkHandling(): void {
    NetInfo.addEventListener(this.handleNetworkChange);
  }

  private handleAppStateChange = (nextAppState: AppStateStatus): void => {
    const previousState = this.appState;
    this.appState = nextAppState;

    // Handle app state transitions
    if (previousState === 'background' && nextAppState === 'active') {
      // App came to foreground
```

```
    this.handleAppForegrounded();
  } else if (previousState === 'active' && nextAppState === 'background') {
    // App went to background
    this.handleAppBackgrounded();
  }
};

private handleNetworkChange = (state: any): void => {
  const wasConnected = this.isNetworkConnected;
  this.isNetworkConnected = state.isConnected;

  if (!wasConnected && state.isConnected) {
    // Network reconnected
    console.log('Network reconnected, attempting socket reconnection');
    this.connect();
  } else if (wasConnected && !state.isConnected) {
    // Network disconnected
    console.log('Network disconnected');
    this.disconnect();
  }
};

private handleAppForegrounded(): void {
  console.log('App foregrounded, checking socket connection');

  // Reconnect if needed
  if (this.connectionState === 'disconnected' && this.isNetworkConnected) {
    this.connect();
  }

  // Start heartbeat
  this.startHeartbeat();

  // Process queued messages
  this.processMessageQueue();
}

private handleAppBackgrounded(): void {
  console.log('App backgrounded');

  // Stop heartbeat to save battery
  this.stopHeartbeat();

  // Optionally disconnect to save battery
```

```
    // this.disconnect();
  }

  async connect(): Promise<void> {
    if (this.connectionState === 'connected' || this.connectionState === 'connecting') {
      return;
    }

    if (!this.isNetworkConnected) {
      console.log('No network connection, queueing socket connection');
      return;
    }

    this.connectionState = 'connecting';

    try {
      const token = await this.getAuthToken();
      const userId = useUserStore.getState().user?.id;

      this.socket = io(process.env.EXPO_PUBLIC_WEBSOCKET_URL || 'ws://localhost:8080', {
        auth: { token, userId },
        transports: ['websocket'],
        timeout: this.options.timeout || 20000,
        reconnection: this.options.reconnection ?? true,
        reconnectionAttempts: this.options.reconnectionAttempts ?? 5,
        reconnectionDelay: this.options.reconnectionDelay ?? 1000,
        autoConnect: this.options.autoConnect ?? true,
      });

      this.setupSocketEventHandlers();

    } catch (error) {
      console.error('Failed to connect socket:', error);
      this.connectionState = 'disconnected';
    }
  }

  private async getAuthToken(): Promise<string | null> {
    try {
      return await AsyncStorage.getItem('auth_token');
    } catch (error) {
      console.error('Failed to get auth token:', error);
      return null;
    }
```

```
}

private setupSocketEventHandlers(): void {
  if (!this.socket) return;

  this.socket.on('connect', () => {
    console.log('Socket connected');
    this.connectionState = 'connected';
    this.startHeartbeat();
    this.processMessageQueue();
    this.notifyListeners('connection', { status: 'connected' });
  });

  this.socket.on('disconnect', (reason) => {
    console.log('Socket disconnected:', reason);
    this.connectionState = 'disconnected';
    this.stopHeartbeat();
    this.notifyListeners('connection', { status: 'disconnected', reason });
  });

  this.socket.on('connect_error', (error) => {
    console.error('Socket connection error:', error);
    this.connectionState = 'disconnected';
    this.notifyListeners('connection', { status: 'error', error });
  });

  this.socket.on('reconnect', (attemptNumber) => {
    console.log('Socket reconnected after', attemptNumber, 'attempts');
    this.connectionState = 'connected';
    this.notifyListeners('connection', { status: 'reconnected', attempts: attemptNumber });
  });

  this.socket.on('reconnect_error', (error) => {
    console.error('Socket reconnection error:', error);
  });

  // Application-specific event handlers
  this.setupApplicationEventHandlers();
}

private setupApplicationEventHandlers(): void {
  if (!this.socket) return;

  // Event updates
```

```
    this.socket.on('event:update', (data) => {
      this.notifyListeners('event:update', data);
    });

    // New notifications
    this.socket.on('notification:new', (data) => {
      this.notifyListeners('notification:new', data);
      // Show local notification if app is in background
      if (this.appState !== 'active') {
        this.showLocalNotification(data);
      }
    });

    // User status updates
    this.socket.on('user:status', (data) => {
      this.notifyListeners('user:status', data);
    });

    // Real-time chat messages
    this.socket.on('message:new', (data) => {
      this.notifyListeners('message:new', data);
    });

    // Location updates from other users
    this.socket.on('location:update', (data) => {
      this.notifyListeners('location:update', data);
    });
  }

  private startHeartbeat(): void {
    this.stopHeartbeat();

    this.heartbeatInterval = setInterval(() => {
      if (this.socket?.connected) {
        this.socket.emit('heartbeat', { timestamp: Date.now() });
      }
    }, 30000); // 30 seconds
  }

  private stopHeartbeat(): void {
    if (this.heartbeatInterval) {
      clearInterval(this.heartbeatInterval);
      this.heartbeatInterval = null;
    }
```

```
}

private processMessageQueue(): void {
  if (this.connectionState !== 'connected' || this.messageQueue.length === 0) {
    return;
  }

  console.log(`Processing ${this.messageQueue.length} queued messages`);

  while (this.messageQueue.length > 0) {
    const message = this.messageQueue.shift();
    if (message) {
      this.emit(message.event, message.data);
    }
  }
}

private showLocalNotification(data: any): void {
  // Implementation depends on local notification library
  // For example, using @react-native-async-storage/async-storage
  console.log('Showing local notification:', data);
}

emit(event: string, data: any): void {
  if (this.connectionState === 'connected' && this.socket) {
    this.socket.emit(event, data);
  } else {
    // Queue message for later
    this.messageQueue.push({ event, data });
    console.log(`Queued message: ${event}`, data);
  }
}

on(event: string, callback: Function): void {
  if (!this.listeners.has(event)) {
    this.listeners.set(event, []);
  }
  this.listeners.get(event)!.push(callback);
}

off(event: string, callback?: Function): void {
  if (!this.listeners.has(event)) return;

  if (callback) {
```

```
      const callbacks = this.listeners.get(event)!;
      const index = callbacks.indexOf(callback);
      if (index !== -1) {
        callbacks.splice(index, 1);
      }
    } else {
      this.listeners.delete(event);
    }
  }

  private notifyListeners(event: string, data: any): void {
    const callbacks = this.listeners.get(event);
    if (callbacks) {
      callbacks.forEach(callback => callback(data));
    }
  }

  disconnect(): void {
    this.stopHeartbeat();

    if (this.socket) {
      this.socket.disconnect();
      this.socket = null;
    }

    this.connectionState = 'disconnected';
  }

  getConnectionState(): string {
    return this.connectionState;
  }

  isConnected(): boolean {
    return this.connectionState === 'connected';
  }

  cleanup(): void {
    this.disconnect();
    AppState.removeEventListener('change', this.handleAppStateChange);
    // NetInfo cleanup would go here
    this.listeners.clear();
  }
}
```

```
export const mobileSocketClient = new MobileSocketClient();
```

2.2 React Hooks for WebSocket

typescript

```typescript
// hooks/useSocket.ts
import { useEffect, useState, useCallback, useRef } from 'react';
import { AppState } from 'react-native';
import { mobileSocketClient } from '@/services/websocket/MobileSocketClient';

interface UseSocketOptions {
  autoConnect?: boolean;
  dependencies?: any[];
}

export const useSocket = (options: UseSocketOptions = {}) => {
  const [isConnected, setIsConnected] = useState(false);
  const [connectionError, setConnectionError] = useState<string | null>(null);
  const [lastActivity, setLastActivity] = useState<Date | null>(null);
  const appState = useRef(AppState.currentState);

  useEffect(() => {
    const handleConnectionChange = (data: any) => {
      setIsConnected(data.status === 'connected' || data.status === 'reconnected');
      setConnectionError(data.status === 'error' ? data.error?.message : null);
      setLastActivity(new Date());
    };

    mobileSocketClient.on('connection', handleConnectionChange);

    if (options.autoConnect !== false) {
      mobileSocketClient.connect();
    }

    return () => {
      mobileSocketClient.off('connection', handleConnectionChange);
    };
  }, options.dependencies || []);

  const emit = useCallback((event: string, data: any) => {
    mobileSocketClient.emit(event, data);
  }, []);

  const subscribe = useCallback((event: string, callback: Function) => {
    mobileSocketClient.on(event, callback);
    return () => mobileSocketClient.off(event, callback);
  }, []);
```

```
  return {
    isConnected,
    connectionError,
    lastActivity,
    emit,
    subscribe,
    connect: () => mobileSocketClient.connect(),
    disconnect: () => mobileSocketClient.disconnect(),
  };
};

// Specific hooks for different features
export const useEventUpdates = () => {
  const [events, setEvents] = useState<any[]>([]);
  const { subscribe } = useSocket();

  useEffect(() => {
    const unsubscribe = subscribe('event:update', (data: any) => {
      setEvents(prev => {
        const index = prev.findIndex(event => event.id === data.event.id);
        if (index !== -1) {
          const updated = [...prev];
          updated[index] = data.event;
          return updated;
        }
        return [...prev, data.event];
      });
    });

    return unsubscribe;
  }, [subscribe]);

  return events;
};

export const useNotifications = () => {
  const [notifications, setNotifications] = useState<any[]>([]);
  const { subscribe } = useSocket();

  useEffect(() => {
    const unsubscribe = subscribe('notification:new', (notification: any) => {
      setNotifications(prev => [notification, ...prev]);
    });
```

```
    return unsubscribe;
  }, [subscribe]);

  const markAsRead = useCallback((notificationId: string) => {
    setNotifications(prev =>
      prev.map(notif =>
        notif.id === notificationId
          ? { ...notif, read: true }
          : notif
      )
    );
  }, []);

  const clearAll = useCallback(() => {
    setNotifications([]);
  }, []);

  return {
    notifications,
    markAsRead,
    clearAll,
    unreadCount: notifications.filter(n => !n.read).length,
  };
};

export const useRealTimeLocation = (eventId: string) => {
  const [userLocations, setUserLocations] = useState<Map<string, any>>(new Map());
  const { subscribe, emit } = useSocket();

  useEffect(() => {
    // Subscribe to location updates for this event
    const unsubscribe = subscribe('location:update', (data: any) => {
      if (data.eventId === eventId) {
        setUserLocations(prev => {
          const updated = new Map(prev);
          updated.set(data.userId, {
            ...data.location,
            timestamp: new Date(data.timestamp),
          });
          return updated;
        });
      }
    });
```

```
    return unsubscribe;
  }, [eventId, subscribe]);

  const shareLocation = useCallback((location: any) => {
    emit('location:share', {
      eventId,
      location,
      timestamp: new Date().toISOString(),
    });
  }, [eventId, emit]);

  const stopSharing = useCallback(() => {
    emit('location:stop_sharing', { eventId });
  }, [eventId, emit]);

  return {
    userLocations,
    shareLocation,
    stopSharing,
  };
};
```

2.3 WebSocket Event Handling Patterns

typescript

```typescript
// services/websocket/EventHandlers.ts
import { queryClient } from '@/services/api/queryClient';
import { useUserStore } from '@/stores/userStore';
import { useUIStore } from '@/stores/uiStore';
import { pushNotificationService } from '@/services/notifications/PushNotificationService';

export class SocketEventHandlers {
  static handleEventUpdate = (data: any) => {
    // Update React Query cache
    queryClient.setQueryData(['events', data.eventId], (oldData: any) => {
      if (!oldData) return data.event;
      return { ...oldData, ...data.event };
    });

    // Invalidate related queries
    queryClient.invalidateQueries(['events']);
    queryClient.invalidateQueries(['user-events']);

    // Show notification if user is affected
    const userStore = useUserStore.getState();
    if (data.affectedUsers?.includes(userStore.user?.id)) {
      const uiStore = useUIStore.getState();
      uiStore.addNotification({
        id: `event-update-${data.eventId}`,
        title: 'Event Updated',
        message: `"${data.event.title}" has been updated`,
        type: 'info',
        timestamp: new Date(),
      });
    }

    console.log('Event updated:', data);
  };

  static handleNewNotification = (notification: any) => {
    const uiStore = useUIStore.getState();

    // Add to UI store
    uiStore.addNotification({
      ...notification,
      timestamp: new Date(notification.timestamp),
    });
```

```
    // Show local push notification if app is backgrounded
    if (AppState.currentState !== 'active') {
      pushNotificationService.showLocalNotification({
        title: notification.title,
        body: notification.message,
        data: notification.data,
      });
    }

    // Play notification sound or haptic feedback
    if (AppState.currentState === 'active') {
      // Haptic feedback for iOS
      if (Platform.OS === 'ios') {
        const { ImpactFeedbackGenerator } = require('expo-haptics');
        ImpactFeedbackGenerator.impactAsync(
          ImpactFeedbackGenerator.ImpactFeedbackStyle.Light
        );
      }
    }

    console.log('New notification:', notification);
  };

  static handleUserStatusUpdate = (data: any) => {
    // Update user presence in cache
    queryClient.setQueryData(['user-presence'], (oldData: any) => {
      if (!oldData) return { [data.userId]: data.status };
      return { ...oldData, [data.userId]: data.status };
    });

    // Update event participants if relevant
    queryClient.setQueriesData(['events'], (oldData: any) => {
      if (!oldData?.participants) return oldData;

      const updatedParticipants = oldData.participants.map((participant: any) =>
        participant.id === data.userId
          ? { ...participant, status: data.status, lastSeen: data.lastSeen }
          : participant
      );

      return { ...oldData, participants: updatedParticipants };
    });

    console.log('User status updated:', data);
```

```
};

static handleNewMessage = (data: any) => {
 // Update messages cache
 queryClient.setQueryData(['messages', data.conversationId], (oldData: any) => {
  if (!oldData) return [data.message];
  return [...oldData, data.message];
 });

 // Update conversation list
 queryClient.setQueryData(['conversations'], (oldData: any) => {
  if (!oldData) return oldData;

  return oldData.map((conversation: any) =>
   conversation.id === data.conversationId
    ? {
       ...conversation,
       lastMessage: data.message,
       updatedAt: data.message.timestamp,
       unreadCount: conversation.unreadCount + 1,
      }
    : conversation
  );
 });

 // Show notification
 const userStore = useUserStore.getState();
 if (data.message.senderId !== userStore.user?.id) {
  const uiStore = useUIStore.getState();
  uiStore.addNotification({
   id: `message-${data.message.id}`,
   title: data.message.senderName,
   message: data.message.content,
   type: 'message',
   timestamp: new Date(data.message.timestamp),
   data: { conversationId: data.conversationId },
  });
 }

 console.log('New message:', data);
};

static handleLocationUpdate = (data: any) => {
 // Update location cache
```

```
    queryClient.setQueryData(['event-locations', data.eventId], (oldData: any) => {
      if (!oldData) return { [data.userId]: data.location };
      return { ...oldData, [data.userId]: data.location };
    });

    // Update user location in event participants
    queryClient.setQueryData(['events', data.eventId], (oldData: any) => {
      if (!oldData?.participants) return oldData;

      const updatedParticipants = oldData.participants.map((participant: any) =>
        participant.id === data.userId
          ? { ...participant, location: data.location, locationUpdatedAt: data.timestamp }
          : participant
      );

      return { ...oldData, participants: updatedParticipants };
    });

    console.log('Location updated:', data);
  };

  static handleConnectionLost = () => {
    const uiStore = useUIStore.getState();
    uiStore.addNotification({
      id: 'connection-lost',
      title: 'Connection Lost',
      message: 'Attempting to reconnect...',
      type: 'warning',
      timestamp: new Date(),
      persistent: true,
    });
  };

  static handleConnectionRestored = () => {
    const uiStore = useUIStore.getState();

    // Remove connection lost notification
    uiStore.removeNotification('connection-lost');

    // Show connection restored notification
    uiStore.addNotification({
      id: 'connection-restored',
      title: 'Connection Restored',
      message: 'You are now connected',
```

```
      type: 'success',
      timestamp: new Date(),
    });

    // Refresh critical data
    queryClient.invalidateQueries(['events']);
    queryClient.invalidateQueries(['notifications']);
    queryClient.invalidateQueries(['conversations']);
  };
}

// Register all event handlers
export const registerSocketEventHandlers = () => {
  mobileSocketClient.on('event:update', SocketEventHandlers.handleEventUpdate);
  mobileSocketClient.on('notification:new', SocketEventHandlers.handleNewNotification);
  mobileSocketClient.on('user:status', SocketEventHandlers.handleUserStatusUpdate);
  mobileSocketClient.on('message:new', SocketEventHandlers.handleNewMessage);
  mobileSocketClient.on('location:update', SocketEventHandlers.handleLocationUpdate);
  mobileSocketClient.on('connection', (data: any) => {
    if (data.status === 'disconnected') {
      SocketEventHandlers.handleConnectionLost();
    } else if (data.status === 'connected' || data.status === 'reconnected') {
      SocketEventHandlers.handleConnectionRestored();
    }
  });
};
```

2.4 Background Task Management

typescript

```
// services/background/BackgroundTaskManager.ts
import BackgroundTask from 'react-native-background-task';
import { AppState, AppStateStatus } from 'react-native';
import { mobileSocketClient } from '@/services/websocket/MobileSocketClient';
import AsyncStorage from '@react-native-async-storage/async-storage';

class BackgroundTaskManager {
  private backgroundTaskId: number | null = null;
  private isBackgroundTaskRunning = false;
  private appState: AppStateStatus = 'active';

  constructor() {
    this.setupAppStateHandling();
  }

  private setupAppStateHandling(): void {
    AppState.addEventListener('change', this.handleAppStateChange);
  }

  private handleAppStateChange = (nextAppState: AppStateStatus): void => {
    const previousState = this.appState;
    this.appState = nextAppState;

    if (previousState === 'active' && nextAppState.match(/inactive|background/)) {
      // App is going to background
      this.startBackgroundTask();
    } else if (previousState.match(/inactive|background/) && nextAppState === 'active') {
      // App is coming to foreground
      this.stopBackgroundTask();
    }
  };

  private startBackgroundTask(): void {
    if (this.isBackgroundTaskRunning) return;

    console.log('Starting background task');

    BackgroundTask.define(() => {
      this.isBackgroundTaskRunning = true;

      // Keep WebSocket connection alive with reduced frequency
      const backgroundInterval = setInterval(() => {
        if (mobileSocketClient.isConnected()) {
```

```
        mobileSocketClient.emit('heartbeat', {
          timestamp: Date.now(),
          background: true
        });
      }
    }, 60000); // Every minute instead of 30 seconds

    // Handle background messages with reduced processing
    const handleBackgroundMessage = (data: any) => {
      // Only process high-priority notifications
      if (data.priority === 'high' || data.type === 'emergency') {
        this.processHighPriorityMessage(data);
      } else {
        // Queue for when app becomes active
        this.queueMessage(data);
      }
    };

    mobileSocketClient.on('notification:new', handleBackgroundMessage);
    mobileSocketClient.on('message:new', handleBackgroundMessage);

    // Cleanup when task ends
    BackgroundTask.finish(() => {
      clearInterval(backgroundInterval);
      mobileSocketClient.off('notification:new', handleBackgroundMessage);
      mobileSocketClient.off('message:new', handleBackgroundMessage);
      this.isBackgroundTaskRunning = false;
      console.log('Background task finished');
    });
  });

  BackgroundTask.start();
}

private stopBackgroundTask(): void {
  if (this.backgroundTaskId) {
    BackgroundTask.stop();
    this.backgroundTaskId = null;
  }

  // Process queued messages
  this.processQueuedMessages();
}
```

```
private async processHighPriorityMessage(data: any): Promise<void> {
  // Show local notification immediately
  if (data.type === 'emergency') {
    // Handle emergency notifications
    await this.showEmergencyNotification(data);
  } else {
    // Handle high-priority notifications
    await this.showHighPriorityNotification(data);
  }
}

private async queueMessage(data: any): Promise<void> {
  try {
    const queuedMessages = await AsyncStorage.getItem('queued_messages');
    const messages = queuedMessages ? JSON.parse(queuedMessages) : [];
    messages.push({ ...data, queuedAt: Date.now() });

    // Keep only last 50 messages to avoid storage bloat
    const recentMessages = messages.slice(-50);
    await AsyncStorage.setItem('queued_messages', JSON.stringify(recentMessages));
  } catch (error) {
    console.error('Failed to queue message:', error);
  }
}

private async processQueuedMessages(): Promise<void> {
  try {
    const queuedMessages = await AsyncStorage.getItem('queued_messages');
    if (queuedMessages) {
      const messages = JSON.parse(queuedMessages);

      // Process each queued message
      messages.forEach((message: any) => {
        // Only process messages from last 10 minutes to avoid spam
        if (Date.now() - message.queuedAt < 600000) {
          this.processMessage(message);
        }
      });

      // Clear processed messages
      await AsyncStorage.removeItem('queued_messages');
    }
  } catch (error) {
    console.error('Failed to process queued messages:', error);
```

```
  }
 }

 private processMessage(message: any): void {
  // Process message based on type
  switch (message.type) {
   case 'notification':
    SocketEventHandlers.handleNewNotification(message);
    break;
   case 'message':
    SocketEventHandlers.handleNewMessage(message);
    break;
   case 'event_update':
    SocketEventHandlers.handleEventUpdate(message);
    break;
   default:
    console.log('Unknown message type:', message.type);
  }
 }

 private async showEmergencyNotification(data: any): Promise<void> {
  // Implementation for emergency notifications
  // This might include sound, vibration, and persistent notification
  console.log('Emergency notification:', data);
 }

 private async showHighPriorityNotification(data: any): Promise<void> {
  // Implementation for high-priority notifications
  console.log('High-priority notification:', data);
 }

 cleanup(): void {
  this.stopBackgroundTask();
  AppState.removeEventListener('change', this.handleAppStateChange);
 }
}

export const backgroundTaskManager = new BackgroundTaskManager();
```

2.5 Offline Synchronization

typescript

```typescript
// services/sync/OfflineSyncManager.ts
import AsyncStorage from '@react-native-async-storage/async-storage';
import NetInfo from '@react-native-community/netinfo';
import { mobileSocketClient } from '@/services/websocket/MobileSocketClient';
import { apiClient } from '@/services/api/client';

interface SyncItem {
  id: string;
  type: 'create' | 'update' | 'delete';
  entity: string;
  data: any;
  timestamp: number;
  retryCount: number;
}

class OfflineSyncManager {
  private syncQueue: SyncItem[] = [];
  private isOnline = true;
  private isSyncing = false;
  private maxRetries = 3;

  constructor() {
    this.setupNetworkListener();
    this.loadSyncQueue();
  }

  private setupNetworkListener(): void {
    NetInfo.addEventListener(state => {
      const wasOnline = this.isOnline;
      this.isOnline = state.isConnected ?? false;

      if (!wasOnline && this.isOnline) {
        // Just came online, start sync
        console.log('Network restored, starting sync');
        this.syncPendingItems();
      }
    });
  }

  private async loadSyncQueue(): Promise<void> {
    try {
      const queueData = await AsyncStorage.getItem('sync_queue');
      if (queueData) {
```

```
      this.syncQueue = JSON.parse(queueData);
      console.log(`Loaded ${this.syncQueue.length} items from sync queue`);
    }
  } catch (error) {
    console.error('Failed to load sync queue:', error);
  }
}

private async saveSyncQueue(): Promise<void> {
  try {
    await AsyncStorage.setItem('sync_queue', JSON.stringify(this.syncQueue));
  } catch (error) {
    console.error('Failed to save sync queue:', error);
  }
}

async addToQueue(item: Omit<SyncItem, 'id' | 'timestamp' | 'retryCount'>): Promise<void> {
  const syncItem: SyncItem = {
    ...item,
    id: `${item.entity}_${Date.now()}_${Math.random()}`,
    timestamp: Date.now(),
    retryCount: 0,
  };

  this.syncQueue.push(syncItem);
  await this.saveSyncQueue();

  // Try to sync immediately if online
  if (this.isOnline) {
    this.syncPendingItems();
  }
}

private async syncPendingItems(): Promise<void> {
  if (this.isSyncing || !this.isOnline || this.syncQueue.length === 0) {
    return;
  }

  this.isSyncing = true;
  console.log(`Starting sync of ${this.syncQueue.length} items`);

  const itemsToSync = [...this.syncQueue];
  const failedItems: SyncItem[] = [];
```

```
    for (const item of itemsToSync) {
      try {
        await this.syncItem(item);
        console.log(`Successfully synced item: ${item.id}`);

        // Remove from queue
        this.syncQueue = this.syncQueue.filter(queueItem => queueItem.id !== item.id);
      } catch (error) {
        console.error(`Failed to sync item ${item.id}:`, error);

        // Increment retry count
        item.retryCount++;

        if (item.retryCount < this.maxRetries) {
          failedItems.push(item);
        } else {
          console.error(`Max retries exceeded for item: ${item.id}`);
          // Remove from queue after max retries
          this.syncQueue = this.syncQueue.filter(queueItem => queueItem.id !== item.id);
        }
      }
    }

    // Update queue with failed items
    this.syncQueue = [...failedItems];
    await this.saveSyncQueue();

    this.isSyncing = false;
    console.log(`Sync completed. ${this.syncQueue.length} items remaining in queue`);
  }

  private async syncItem(item: SyncItem): Promise<void> {
    switch (item.entity) {
      case 'event':
        return this.syncEvent(item);
      case 'message':
        return this.syncMessage(item);
      case 'user_profile':
        return this.syncUserProfile(item);
      default:
        throw new Error(`Unknown entity type: ${item.entity}`);
    }
  }
```

```
private async syncEvent(item: SyncItem): Promise<void> {
  const endpoint = `/events${item.data.id ? `/${item.data.id}` : ''}`;

  switch (item.type) {
    case 'create':
      await apiClient.post(endpoint, item.data);
      break;
    case 'update':
      await apiClient.put(endpoint, item.data);
      break;
    case 'delete':
      await apiClient.delete(endpoint);
      break;
  }
}

private async syncMessage(item: SyncItem): Promise<void> {
  const endpoint = `/messages${item.data.id ? `/${item.data.id}` : ''}`;

  switch (item.type) {
    case 'create':
      await apiClient.post(endpoint, item.data);
      // Emit via socket if connected
      if (mobileSocketClient.isConnected()) {
        mobileSocketClient.emit('message:send', item.data);
      }
      break;
    case 'update':
      await apiClient.put(endpoint, item.data);
      break;
    case 'delete':
      await apiClient.delete(endpoint);
      break;
  }
}

private async syncUserProfile(item: SyncItem): Promise<void> {
  const endpoint = `/users/profile`;

  switch (item.type) {
    case 'update':
      await apiClient.put(endpoint, item.data);
      break;
    default:
```

```
      throw new Error(`Unsupported operation for user_profile: ${item.type}`);
    }
  }

  // Public methods for adding operations to sync queue
  async queueEventCreate(eventData: any): Promise<void> {
    return this.addToQueue({
      type: 'create',
      entity: 'event',
      data: eventData,
    });
  }

  async queueEventUpdate(eventId: string, eventData: any): Promise<void> {
    return this.addToQueue({
      type: 'update',
      entity: 'event',
      data: { ...eventData, id: eventId },
    });
  }

  async queueEventDelete(eventId: string): Promise<void> {
    return this.addToQueue({
      type: 'delete',
      entity: 'event',
      data: { id: eventId },
    });
  }

  async queueMessageSend(messageData: any): Promise<void> {
    return this.addToQueue({
      type: 'create',
      entity: 'message',
      data: messageData,
    });
  }

  async queueProfileUpdate(profileData: any): Promise<void> {
    return this.addToQueue({
      type: 'update',
      entity: 'user_profile',
      data: profileData,
    });
  }
```

```
  // Get queue status
  getQueueStatus(): { count: number; isOnline: boolean; isSyncing: boolean } {
    return {
      count: this.syncQueue.length,
      isOnline: this.isOnline,
      isSyncing: this.isSyncing,
    };
  }

  // Force sync (useful for manual retry)
  forcSync(): Promise<void> {
    return this.syncPendingItems();
  }

  // Clear queue (use with caution)
  async clearQueue(): Promise<void> {
    this.syncQueue = [];
    await this.saveSyncQueue();
  }
}

export const offlineSyncManager = new OfflineSyncManager();

// Hook for using offline sync
export const useOfflineSync = () => {
  const [queueStatus, setQueueStatus] = useState(offlineSyncManager.getQueueStatus());

  useEffect(() => {
    const interval = setInterval(() => {
      setQueueStatus(offlineSyncManager.getQueueStatus());
    }, 1000);

    return () => clearInterval(interval);
  }, []);

  return {
    queueStatus,
    queueEventCreate: offlineSyncManager.queueEventCreate.bind(offlineSyncManager),
    queueEventUpdate: offlineSyncManager.queueEventUpdate.bind(offlineSyncManager),
    queueEventDelete: offlineSyncManager.queueEventDelete.bind(offlineSyncManager),
    queueMessageSend: offlineSyncManager.queueMessageSend.bind(offlineSyncManager),
    queueProfileUpdate: offlineSyncManager.queueProfileUpdate.bind(offlineSyncManager),
    forceSync: offlineSyncManager.forcSync.bind(offlineSyncManager),
```

```
  };
};
```

3. Integration Examples

3.1 Real-time Event Screen

typescript

```tsx
// screens/EventDetailScreen.tsx
import React, { useEffect, useState } from 'react';
import { View, Text, ScrollView, Alert } from 'react-native';
import { useRoute, useNavigation } from '@react-navigation/native';
import { useSocket, useRealTimeLocation, useOfflineSync } from '@/hooks';
import { locationService } from '@/native/LocationModule/LocationModule';

export const EventDetailScreen: React.FC = () => {
  const route = useRoute();
  const navigation = useNavigation();
  const { eventId } = route.params as { eventId: string };

  const { isConnected, emit, subscribe } = useSocket();
  const { userLocations, shareLocation, stopSharing } = useRealTimeLocation(eventId);
  const { queueEventUpdate } = useOfflineSync();

  const [event, setEvent] = useState(null);
  const [isSharing, setIsSharing] = useState(false);
  const [participants, setParticipants] = useState([]);

  useEffect(() => {
    // Subscribe to real-time event updates
    const unsubscribeEventUpdate = subscribe('event:update', (data: any) => {
      if (data.eventId === eventId) {
        setEvent(data.event);
      }
    });

    const unsubscribeParticipantUpdate = subscribe('participant:update', (data: any) => {
      if (data.eventId === eventId) {
        setParticipants(prev => {
          const index = prev.findIndex(p => p.id === data.participant.id);
          if (index !== -1) {
            const updated = [...prev];
            updated[index] = data.participant;
            return updated;
          }
          return [...prev, data.participant];
        });
      }
    });

    return () => {
```

```
    unsubscribeEventUpdate();
    unsubscribeParticipantUpdate();
  };
}, [eventId, subscribe]);

const handleStartLocationSharing = async () => {
  try {
    const hasPermission = await locationService.requestPermission();
    if (!hasPermission) {
      Alert.alert('Permission Required', 'Location permission is needed to share your location.');
      return;
    }

    const currentLocation = await locationService.getCurrentLocation();
    shareLocation(currentLocation);
    setIsSharing(true);

    // Start watching location changes
    locationService.startWatchingLocation(
      (location) => {
        shareLocation(location);
      },
      (error) => {
        console.error('Location error:', error);
        Alert.alert('Location Error', 'Failed to get your location.');
      }
    );
  } catch (error) {
    console.error('Failed to start location sharing:', error);
    Alert.alert('Error', 'Failed to start location sharing.');
  }
};

const handleStopLocationSharing = () => {
  locationService.stopWatchingLocation();
  stopSharing();
  setIsSharing(false);
};

const handleJoinEvent = () => {
  if (isConnected) {
    emit('event:join', { eventId });
  } else {
    // Queue for offline sync
```

```
    queueEventUpdate(eventId, { joined: true });
  }
};

return (
  <ScrollView style={styles.container}>
    <View style={styles.connectionStatus}>
      <Text style={[styles.statusText, { color: isConnected ? 'green' : 'orange' }]}>
        {isConnected ? 'Connected' : 'Offline'}
      </Text>
    </View>

    {event && (
      <View style={styles.eventDetails}>
        <Text style={styles.title}>{event.title}</Text>
        <Text style={styles.description}>{event.description}</Text>

        <View style={styles.participantsList}>
          <Text style={styles.sectionTitle}>Participants ({participants.length})</Text>
          {participants.map(participant => (
            <View key={participant.id} style={styles.participantItem}>
              <Text>{participant.name}</Text>
              {userLocations.has(participant.id) && (
                <Text style={styles.locationIndicator}>📍 Location shared</Text>
              )}
            </View>
          ))}
        </View>

        <View style={styles.actions}>
          <Button
            title={isSharing ? "Stop Sharing Location" : "Share Location"}
            onPress={isSharing ? handleStopLocationSharing : handleStartLocationSharing}
          />
          <Button
            title="Join Event"
            onPress={handleJoinEvent}
          />
        </View>
      </View>
    )}
  </ScrollView>
```

```
  );
};
```

This comprehensive deep dive covers the essential mobile-specific patterns and WebSocket integration strategies for your React Native applications. The architecture provides robust offline support, real-time communication, background task management, and seamless synchronization with your backend microservices.