dart file:<main.dart>

```dart
// lib/main.dart
// ============================================
// POINT D'ENTRÉE DE L'APPLICATION
// ============================================

import 'package:flutter/material.dart';
import 'package:supabase_flutter/supabase_flutter.dart';
import 'package:intl/date_symbol_data_local.dart';
import 'package:flutter_localizations/flutter_localizations.dart';
import 'utils/constants.dart';
import 'utils/theme.dart';
import 'services/auth_service.dart';
import 'services/notification_service.dart';
import 'screens/login_screen.dart';
import 'screens/home_screen.dart';

void main() async {
  // S'assurer que Flutter est initialisé
  WidgetsFlutterBinding.ensureInitialized();

  // Initialiser les formats de date en français
  await initializeDateFormatting('fr_FR', null);

  // Initialiser Supabase
  await Supabase.initialize(
    url: AppConstants.supabaseUrl,
    anonKey: AppConstants.supabaseAnonKey,
  );

  // Initialiser les notifications locales
  await NotificationService().initialize();

  // Vérifier s'il y a une session active
  final authService = AuthService();
  final savedUser = await authService.restoreSession();

  runApp(MyApp(isLoggedIn: savedUser != null));
}

class MyApp extends StatelessWidget {
  final bool isLoggedIn;

  const MyApp({super.key, required this.isLoggedIn});

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Entretien Immeuble',
      debugShowCheckedModeBanner: false,
      theme: AppTheme.lightTheme,

      // ============================================
      // LOCALISATIONS — C'EST CECI QUI CORRIGE LE CALENDRIER
      // ============================================
      locale: const Locale('fr', 'FR'),
      supportedLocales: const [
        Locale('fr', 'FR'),
        Locale('en', 'US'),
      ],
      localizationsDelegates: const [
        GlobalMaterialLocalizations.delegate,
        GlobalWidgetsLocalizations.delegate,
        GlobalCupertinoLocalizations.delegate,
      ],

      home: isLoggedIn ? const HomeScreen() : const LoginScreen(),
    );
  }
}
```

dart file:<immeuble_model.dart>

```dart
// lib/models/immeuble_model.dart
// ============================================
// MODÈLE DE DONNÉES POUR UN IMMEUBLE
// ============================================

class ImmeubleModel {
  final String id;
  final String nom;
  final String adresse;
  final bool archived;
  final DateTime createdAt;

  ImmeubleModel({
    required this.id,
    required this.nom,
    this.adresse = '',
    this.archived = false,
    DateTime? createdAt,
  }) : createdAt = createdAt ?? DateTime.now();

  factory ImmeubleModel.fromMap(Map<String, dynamic> map) {
    return ImmeubleModel(
      id: map['id'] ?? '',
      nom: map['nom'] ?? '',
      adresse: map['adresse'] ?? '',
      archived: map['archived'] == true || map['archived'] == 1,
```

```
      createdAt: map['created_at'] != null
          ? DateTime.tryParse(map['created_at'].toString()) ?? DateTime.now()
          : DateTime.now(),
    );
  }

  Map<String, dynamic> toMapSupabase() {
    return {
      'id': id,
      'nom': nom,
      'adresse': adresse,
      'archived': archived,
      'created_at': createdAt.toIso8601String(),
    };
  }

  Map<String, dynamic> toMapLocal() {
    return {
      'id': id,
      'nom': nom,
      'adresse': adresse,
      'archived': archived ? 1 : 0,
      'created_at': createdAt.toIso8601String(),
    };
  }
}


***dart file:<task_history_model.dart>***

// lib/models/task_history_model.dart
// ================================================
// MODÈLE POUR L'HISTORIQUE DES MODIFICATIONS
// ================================================

class TaskHistoryModel {
  final int? id;
  final int? serverId;
  final String taskId;
  final String champModifie;
  final String ancienneValeur;
  final String nouvelleValeur;
  final String modifiedBy;
  final String modifiedByName;
  final DateTime modifiedAt;
  final String syncStatus;

  TaskHistoryModel({
    this.id,
    this.serverId,
    required this.taskId,
    required this.champModifie,
    this.ancienneValeur = '',
    this.nouvelleValeur = '',
    this.modifiedBy = '',
    this.modifiedByName = '',
    DateTime? modifiedAt,
    this.syncStatus = 'synced',
  }) : modifiedAt = modifiedAt ?? DateTime.now();

  factory TaskHistoryModel.fromMap(Map<String, dynamic> map) {
    return TaskHistoryModel(
      id: map['id'] is int
          ? map['id']
          : int.tryParse(map['id']?.toString() ?? ''),
      serverId: map['server_id'] is int
          ? map['server_id']
          : int.tryParse(map['server_id']?.toString() ?? ''),
      taskId: map['task_id'] ?? '',
      champModifie: map['champ_modifie'] ?? '',
      ancienneValeur: map['ancienne_valeur'] ?? '',
      nouvelleValeur: map['nouvelle_valeur'] ?? '',
      modifiedBy: map['modified_by'] ?? '',
      modifiedByName: map['modified_by_name'] ?? '',
      modifiedAt: map['modified_at'] != null
          ? DateTime.tryParse(map['modified_at'].toString()) ?? DateTime.now()
          : DateTime.now(),
      syncStatus: map['sync_status'] ?? 'synced',
    );
  }

  Map<String, dynamic> toMapSupabase() {
    return {
      'task_id': taskId,
      'champ_modifie': champModifie,
      'ancienne_valeur': ancienneValeur,
      'nouvelle_valeur': nouvelleValeur,
      'modified_by': modifiedBy,
      'modified_by_name': modifiedByName,
      'modified_at': modifiedAt.toIso8601String(),
    };
  }

  Map<String, dynamic> toMapLocal() {
    return {
      'server_id': serverId,
      'task_id': taskId,
      'champ_modifie': champModifie,
      'ancienne_valeur': ancienneValeur,
      'nouvelle_valeur': nouvelleValeur,
      'modified_by': modifiedBy,
```

```
      'modified_by_name': modifiedByName,
      'modified_at': modifiedAt.toIso8601String(),
      'sync_status': syncStatus,
    };
  }

  // Nom lisible du champ modifié
  String get champLabel {
    switch (champModifie) {
      case 'immeuble':
        return 'Immeuble';
      case 'etage':
        return 'Étage';
      case 'chambre':
        return 'Chambre';
      case 'description':
        return 'Description';
      case 'done':
        return 'Statut';
      case 'done_date':
        return 'Date d\'exécution';
      case 'done_by':
        return 'Exécutant';
      case 'photo_url':
        return 'Photo';
      case 'archived':
        return 'Archivage';
      case 'planned_date':
        return 'Date planifiée';
      default:
        return champModifie;
    }
  }
}
```

dart file:<task_model.dart>

```
// lib/models/task_model.dart
// ================================================
// MODÈLE DE DONNÉES POUR UNE TÂCHE
// ================================================

class TaskModel {
  final String id;
  final int? taskNumber;
  final DateTime createdAt;
  final DateTime updatedAt;
  final String immeuble;
  final String etage;
  final String chambre;
  final String description;
  final bool done;
  final DateTime? doneDate;
  final String doneBy;
  final String lastModifiedBy;
  final String photoUrl;
  final String photoLocalPath;
  final bool archived;
  final DateTime? plannedDate;
  final bool deleted;
  final String syncStatus;

  TaskModel({
    required this.id,
    this.taskNumber,
    DateTime? createdAt,
    DateTime? updatedAt,
    required this.immeuble,
    this.etage = '',
    this.chambre = '',
    required this.description,
    this.done = false,
    this.doneDate,
    this.doneBy = '',
    this.lastModifiedBy = '',
    this.photoUrl = '',
    this.photoLocalPath = '',
    this.archived = false,
    this.plannedDate,
    this.deleted = false,
    this.syncStatus = 'synced',
  })  : createdAt = createdAt ?? DateTime.now(),
        updatedAt = updatedAt ?? DateTime.now();

  factory TaskModel.fromMap(Map<String, dynamic> map) {
    return TaskModel(
      id: map['id'] ?? '',
      taskNumber: map['task_number'] != null
          ? int.tryParse(map['task_number'].toString())
          : null,
      createdAt: map['created_at'] != null
          ? DateTime.tryParse(map['created_at'].toString()) ?? DateTime.now()
          : DateTime.now(),
      updatedAt: map['updated_at'] != null
          ? DateTime.tryParse(map['updated_at'].toString()) ?? DateTime.now()
          : DateTime.now(),
      immeuble: map['immeuble'] ?? '',
      etage: map['etage'] ?? '',
      chambre: map['chambre'] ?? '',
      description: map['description'] ?? '',
```

```
    done: map['done'] == true || map['done'] == 1,
    doneDate: map['done_date'] != null
        ? DateTime.tryParse(map['done_date'].toString())
        : null,
    doneBy: map['done_by'] ?? '',
    lastModifiedBy: map['last_modified_by'] ?? '',
    photoUrl: map['photo_url'] ?? '',
    photoLocalPath: map['photo_local_path'] ?? '',
    archived: map['archived'] == true || map['archived'] == 1,
    plannedDate: map['planned_date'] != null
        ? DateTime.tryParse(map['planned_date'].toString())
        : null,
    deleted: map['deleted'] == true || map['deleted'] == 1,
    syncStatus: map['sync_status'] ?? 'synced',
  );
}

// Pour Supabase (sans champs locaux)
Map<String, dynamic> toMapSupabase() {
  final map = <String, dynamic>{
    'id': id,
    'created_at': createdAt.toIso8601String(),
    'updated_at': updatedAt.toIso8601String(),
    'immeuble': immeuble,
    'etage': etage,
    'chambre': chambre,
    'description': description,
    'done': done,
    'done_date': doneDate?.toIso8601String(),
    'done_by': doneBy,
    'last_modified_by': lastModifiedBy,
    'photo_url': photoUrl,
    'archived': archived,
    'planned_date': plannedDate?.toIso8601String()?.split('T')[0],
    'deleted': deleted,
  };

  // Inclure le task_number s'il existe
  if (taskNumber != null && taskNumber! > 0) {
    map['task_number'] = taskNumber;
  }

  return map;
}

// Pour SQLite local
Map<String, dynamic> toMapLocal() {
  return {
    'id': id,
    'task_number': taskNumber,
    'created_at': createdAt.toIso8601String(),
    'updated_at': updatedAt.toIso8601String(),
    'immeuble': immeuble,
    'etage': etage,
    'chambre': chambre,
    'description': description,
    'done': done ? 1 : 0,
    'done_date': doneDate?.toIso8601String(),
    'done_by': doneBy,
    'last_modified_by': lastModifiedBy,
    'photo_url': photoUrl,
    'photo_local_path': photoLocalPath,
    'archived': archived ? 1 : 0,
    'planned_date': plannedDate?.toIso8601String()?.split('T')[0],
    'deleted': deleted ? 1 : 0,
    'sync_status': syncStatus,
  };
}

  // Numéro affiché
String get displayNumber {
  if (taskNumber != null && taskNumber! > 0) return '#$taskNumber';
  // Afficher les 6 premiers caractères de l'ID comme numéro temporaire
  return '#${id.substring(0, 6).toUpperCase()}';
}

// Statut sous forme de texte
String get statusText {
  if (archived) return 'Archivée';
  if (done) return 'Terminée';
  return 'En cours';
}

TaskModel copyWith({
  String? id,
  int? taskNumber,
  DateTime? createdAt,
  DateTime? updatedAt,
  String? immeuble,
  String? etage,
  String? chambre,
  String? description,
  bool? done,
  DateTime? doneDate,
  String? doneBy,
  String? lastModifiedBy,
  String? photoUrl,
  String? photoLocalPath,
  bool? archived,
  DateTime? plannedDate,
  bool? deleted,
```

```
    String? syncStatus,
  }) {
    return TaskModel(
      id: id ?? this.id,
      taskNumber: taskNumber ?? this.taskNumber,
      createdAt: createdAt ?? this.createdAt,
      updatedAt: updatedAt ?? DateTime.now(),
      immeuble: immeuble ?? this.immeuble,
      etage: etage ?? this.etage,
      chambre: chambre ?? this.chambre,
      description: description ?? this.description,
      done: done ?? this.done,
      doneDate: doneDate ?? this.doneDate,
      doneBy: doneBy ?? this.doneBy,
      lastModifiedBy: lastModifiedBy ?? this.lastModifiedBy,
      photoUrl: photoUrl ?? this.photoUrl,
      photoLocalPath: photoLocalPath ?? this.photoLocalPath,
      archived: archived ?? this.archived,
      plannedDate: plannedDate ?? this.plannedDate,
      deleted: deleted ?? this.deleted,
      syncStatus: syncStatus ?? this.syncStatus,
    );
  }
}


***dart file:<user_model.dart>***

// lib/models/user_model.dart
// ============================================
// MODÈLE DE DONNÉES POUR UN UTILISATEUR
// ============================================

class UserModel {
  final String id;
  final String identifiant;
  final String motDePasseHash;
  final String nom;
  final String prenom;
  final String telephone;
  final String email;
  final String role;
  final bool archived;
  final DateTime createdAt;
  final DateTime updatedAt;

  UserModel({
    required this.id,
    required this.identifiant,
    required this.motDePasseHash,
    required this.nom,
    required this.prenom,
    this.telephone = '',
    this.email = '',
    required this.role,
    this.archived = false,
    DateTime? createdAt,
    DateTime? updatedAt,
  })  : createdAt = createdAt ?? DateTime.now(),
        updatedAt = updatedAt ?? DateTime.now();

  // Créer depuis un Map (JSON Supabase ou SQLite)
  factory UserModel.fromMap(Map<String, dynamic> map) {
    return UserModel(
      id: map['id'] ?? '',
      identifiant: map['identifiant'] ?? '',
      motDePasseHash: map['mot_de_passe_hash'] ?? '',
      nom: map['nom'] ?? '',
      prenom: map['prenom'] ?? '',
      telephone: map['telephone'] ?? '',
      email: map['email'] ?? '',
      role: map['role'] ?? 'executant',
      archived: map['archived'] == true || map['archived'] == 1,
      createdAt: map['created_at'] != null
          ? DateTime.tryParse(map['created_at'].toString()) ?? DateTime.now()
          : DateTime.now(),
      updatedAt: map['updated_at'] != null
          ? DateTime.tryParse(map['updated_at'].toString()) ?? DateTime.now()
          : DateTime.now(),
    );
  }

  // Convertir en Map pour Supabase
  Map<String, dynamic> toMapSupabase() {
    return {
      'id': id,
      'identifiant': identifiant,
      'mot_de_passe_hash': motDePasseHash,
      'nom': nom,
      'prenom': prenom,
      'telephone': telephone,
      'email': email,
      'role': role,
      'archived': archived,
      'created_at': createdAt.toIso8601String(),
      'updated_at': updatedAt.toIso8601String(),
    };
  }

  // Convertir en Map pour SQLite
  Map<String, dynamic> toMapLocal() {
```

```
    return {
      'id': id,
      'identifiant': identifiant,
      'mot_de_passe_hash': motDePasseHash,
      'nom': nom,
      'prenom': prenom,
      'telephone': telephone,
      'email': email,
      'role': role,
      'archived': archived ? 1 : 0,
      'created_at': createdAt.toIso8601String(),
      'updated_at': updatedAt.toIso8601String(),
    };
  }

  // Nom complet
  String get nomComplet => '$prenom $nom';

  // Vérifier si administrateur
  bool get isAdmin => role == 'administrateur';

  // Copier avec modifications
  UserModel copyWith({
    String? id,
    String? identifiant,
    String? motDePasseHash,
    String? nom,
    String? prenom,
    String? telephone,
    String? email,
    String? role,
    bool? archived,
    DateTime? updatedAt,
  }) {
    return UserModel(
      id: id ?? this.id,
      identifiant: identifiant ?? this.identifiant,
      motDePasseHash: motDePasseHash ?? this.motDePasseHash,
      nom: nom ?? this.nom,
      prenom: prenom ?? this.prenom,
      telephone: telephone ?? this.telephone,
      email: email ?? this.email,
      role: role ?? this.role,
      archived: archived ?? this.archived,
      createdAt: createdAt,
      updatedAt: updatedAt ?? DateTime.now(),
    );
  }
}
```

dart file:<archive_screen.dart>

```
// lib/screens/archive_screen.dart
// ============================================
// ÉCRAN DES TÂCHES ARCHIVÉES
// (Données lues depuis le serveur distant uniquement)
// ============================================
import 'package:flutter/material.dart';
import 'package:intl/intl.dart';
import '../models/task_model.dart';
import '../models/immeuble_model.dart';
import '../services/supabase_service.dart';
import '../services/local_db_service.dart';
import '../services/auth_service.dart';
import '../services/sync_service.dart';
import '../utils/theme.dart';
import '../widgets/app_drawer.dart';
import '../widgets/task_card.dart';
import 'task_detail_screen.dart';

class ArchiveScreen extends StatefulWidget {
  const ArchiveScreen({super.key});

  @override
  State<ArchiveScreen> createState() => _ArchiveScreenState();
}

class _ArchiveScreenState extends State<ArchiveScreen> {
  final SupabaseService _supabase = SupabaseService();
  final AuthService _auth = AuthService();
  final LocalDbService _localDb = LocalDbService();

  List<TaskModel> _archivedTasks = [];
  bool _isLoading = true;
  bool _hasConnection = false;
  String? _errorMessage;

  // Liste des immeubles
  List<ImmeubleModel> _immeubles = [];
  String? _selectedImmeuble;

  // Filtres
  final TextEditingController _etageFilter = TextEditingController();
  final TextEditingController _chambreFilter = TextEditingController();
  final TextEditingController _executantFilter = TextEditingController();
  DateTime? _dateFilter;
  String _sortBy = 'updated_at';
  bool _sortAscending = false;

  @override
```

```
void initState() {
  super.initState();
  _loadImmeubles();
  _checkConnectionAndLoad();
}

@override
void dispose() {
  _etageFilter.dispose();
  _chambreFilter.dispose();
  _executantFilter.dispose();
  super.dispose();
}

Future<void> _loadImmeubles() async {
  final immeubles = await _localDb.getActiveImmeubles();
  if (mounted) {
    setState(() {
      _immeubles = immeubles;
    });
  }
}

Future<void> _checkConnectionAndLoad() async {
  _hasConnection = await SyncService().hasConnection();
  if (_hasConnection) {
    await _loadArchivedTasks();
  } else {
    setState(() {
      _isLoading = false;
      _errorMessage =
          'Pas de connexion internet.\nLes archives sont stockées sur le serveur distant.';
    });
  }
}

Future<void> _loadArchivedTasks() async {
  setState(() {
    _isLoading = true;
    _errorMessage = null;
  });

  try {
    final tasks = await _supabase.getArchivedTasks(
      immeuble: _selectedImmeuble,
      etage: _etageFilter.text.trim().isNotEmpty
          ? _etageFilter.text.trim()
          : null,
      chambre: _chambreFilter.text.trim().isNotEmpty
          ? _chambreFilter.text.trim()
          : null,
      doneBy: _executantFilter.text.trim().isNotEmpty
          ? _executantFilter.text.trim()
          : null,
      doneDate: _dateFilter,
      orderBy: _sortBy,
      ascending: _sortAscending,
    );

    if (mounted) {
      setState(() {
        _archivedTasks = tasks;
        _isLoading = false;
      });
    }
  } catch (e) {
    if (mounted) {
      setState(() {
        _isLoading = false;
        _errorMessage = 'Erreur de chargement: $e';
      });
    }
  }
}

Future<void> _unarchiveTask(TaskModel task) async {
  final confirm = await showDialog<bool>(
    context: context,
    builder: (context) => AlertDialog(
      title: const Text('Désarchiver la tâche ?'),
      content: Text(
          'Voulez-vous désarchiver la tâche ${task.displayNumber} ?\n\n"${task.description}"'),
      actions: [
        TextButton(
          onPressed: () => Navigator.pop(context, false),
          child: const Text('Annuler'),
        ),
        ElevatedButton(
          onPressed: () => Navigator.pop(context, true),
          child: const Text('Désarchiver'),
        ),
      ],
    ),
  );

  if (confirm != true) return;

  try {
    final updatedTask = task.copyWith(
      archived: false,
      lastModifiedBy: _auth.currentUser?.id ?? '',
```

```
      syncStatus: 'synced',
    );

    // 1. Mettre à jour sur le serveur distant
    await _supabase.upsertTask(updatedTask);

    // 2. Réinsérer dans la base locale
    await _localDb.insertTask(updatedTask);

    // 3. Recharger la liste des archives
    await _loadArchivedTasks();

    if (mounted) {
      ScaffoldMessenger.of(context).showSnackBar(
        const SnackBar(
          content: Text('□ Tâche désarchivée et restaurée dans la liste'),
          backgroundColor: AppTheme.successColor,
        ),
      );
    }
  } catch (e) {
    if (mounted) {
      ScaffoldMessenger.of(context).showSnackBar(
        SnackBar(
          content: Text('□ Erreur: $e'),
          backgroundColor: AppTheme.errorColor,
        ),
      );
    }
  }
}

void _showFilterDialog() {
  showModalBottomSheet(
    context: context,
    isScrollControlled: true,
    shape: const RoundedRectangleBorder(
      borderRadius: BorderRadius.vertical(top: Radius.circular(16)),
    ),
    builder: (context) => StatefulBuilder(
      builder: (context, setModalState) => Padding(
        padding: EdgeInsets.only(
          left: 20,
          right: 20,
          top: 20,
          bottom: MediaQuery.of(context).viewInsets.bottom + 20,
        ),
        child: SingleChildScrollView(
          child: Column(
            mainAxisSize: MainAxisSize.min,
            crossAxisAlignment: CrossAxisAlignment.start,
            children: [
              const Text(
                'Filtres et tri',
                style: TextStyle(
                    fontSize: 20, fontWeight: FontWeight.bold),
              ),
              const SizedBox(height: 16),

              // ============================================
              // IMMEUBLE — Liste déroulante
              // ============================================
              DropdownButtonFormField<String>(
                value: _selectedImmeuble,
                decoration: const InputDecoration(
                  labelText: 'Immeuble',
                  prefixIcon: Icon(Icons.apartment),
                ),
                isExpanded: true,
                hint: const Text('Tous les immeubles'),
                items: [
                  const DropdownMenuItem<String>(
                    value: null,
                    child: Text('Tous les immeubles'),
                  ),
                  ..._immeubles.map((immeuble) {
                    return DropdownMenuItem<String>(
                      value: immeuble.nom,
                      child: Text(immeuble.nom),
                    );
                  }),
                ],
                onChanged: (value) {
                  setModalState(() {
                    _selectedImmeuble = value;
                  });
                  setState(() {
                    _selectedImmeuble = value;
                  });
                },
              ),
              const SizedBox(height: 12),

              Row(
                children: [
                  Expanded(
                    child: TextField(
                      controller: _etageFilter,
                      decoration: const InputDecoration(
                        labelText: 'Étage',
                        prefixIcon: Icon(Icons.layers),
```

```
        ),
      ),
    ),
    const SizedBox(width: 12),
    Expanded(
      child: TextField(
        controller: _chambreFilter,
        decoration: const InputDecoration(
          labelText: 'Chambre',
          prefixIcon: Icon(Icons.door_front_door),
        ),
      ),
    ),
  ],
),
const SizedBox(height: 12),
TextField(
  controller: _executantFilter,
  decoration: const InputDecoration(
    labelText: 'Exécutant',
    prefixIcon: Icon(Icons.person),
  ),
),
const SizedBox(height: 12),
ListTile(
  contentPadding: EdgeInsets.zero,
  leading: const Icon(Icons.calendar_today),
  title: Text(
    _dateFilter != null
        ? DateFormat('dd/MM/yyyy').format(_dateFilter!)
        : 'Date d\'exécution',
  ),
  trailing: Row(
    mainAxisSize: MainAxisSize.min,
    children: [
      IconButton(
        icon: const Icon(Icons.edit_calendar),
        onPressed: () async {
          final picked = await showDatePicker(
            context: context,
            initialDate:
                _dateFilter ?? DateTime.now(),
            firstDate: DateTime(2020),
            lastDate: DateTime(2030),
          );
          if (picked != null) {
            setModalState(
                () => _dateFilter = picked);
            setState(
                () => _dateFilter = picked);
          }
        },
      ),
      if (_dateFilter != null)
        IconButton(
          icon: const Icon(Icons.clear,
              color: AppTheme.errorColor),
          onPressed: () {
            setModalState(
                () => _dateFilter = null);
            setState(
                () => _dateFilter = null);
          },
        ),
    ],
  ),
),
const SizedBox(height: 16),
const Text('Trier par :',
    style:
        TextStyle(fontWeight: FontWeight.w600)),
const SizedBox(height: 8),
Wrap(
  spacing: 8,
  children: [
    ChoiceChip(
      label: const Text('Date modif.'),
      selected: _sortBy == 'updated_at',
      onSelected: (_) {
        setModalState(
            () => _sortBy = 'updated_at');
        setState(
            () => _sortBy = 'updated_at');
      },
    ),
    ChoiceChip(
      label: const Text('Immeuble'),
      selected: _sortBy == 'immeuble',
      onSelected: (_) {
        setModalState(
            () => _sortBy = 'immeuble');
        setState(
            () => _sortBy = 'immeuble');
      },
    ),
    ChoiceChip(
      label: const Text('Date exéc.'),
      selected: _sortBy == 'done_date',
      onSelected: (_) {
        setModalState(
            () => _sortBy = 'done_date');
```

```
                    setState(
                        () => _sortBy = 'done_date');
                  },
                ),
                ChoiceChip(
                  label: const Text('Étage'),
                  selected: _sortBy == 'etage',
                  onSelected: (_) {
                    setModalState(
                        () => _sortBy = 'etage');
                    setState(() => _sortBy = 'etage');
                  },
                ),
              ],
            ),
            const SizedBox(height: 8),
            Row(
              children: [
                const Text('Ordre : '),
                ChoiceChip(
                  label: const Text('Croissant ↑'),
                  selected: _sortAscending,
                  onSelected: (_) {
                    setModalState(
                        () => _sortAscending = true);
                    setState(
                        () => _sortAscending = true);
                  },
                ),
                const SizedBox(width: 8),
                ChoiceChip(
                  label: const Text('Décroissant ↓'),
                  selected: !_sortAscending,
                  onSelected: (_) {
                    setModalState(
                        () => _sortAscending = false);
                    setState(
                        () => _sortAscending = false);
                  },
                ),
              ],
            ),
            const SizedBox(height: 20),
            Row(
              children: [
                Expanded(
                  child: OutlinedButton(
                    onPressed: () {
                      _etageFilter.clear();
                      _chambreFilter.clear();
                      _executantFilter.clear();
                      setModalState(() {
                        _selectedImmeuble = null;
                        _dateFilter = null;
                        _sortBy = 'updated_at';
                        _sortAscending = false;
                      });
                      setState(() {
                        _selectedImmeuble = null;
                        _dateFilter = null;
                        _sortBy = 'updated_at';
                        _sortAscending = false;
                      });
                      Navigator.pop(context);
                      _loadArchivedTasks();
                    },
                    child: const Text('Réinitialiser'),
                  ),
                ),
                const SizedBox(width: 12),
                Expanded(
                  child: ElevatedButton(
                    onPressed: () {
                      Navigator.pop(context);
                      _loadArchivedTasks();
                    },
                    child: const Text('Appliquer'),
                  ),
                ),
              ],
            ),
          ],
        ),
      ),
    ),
  );
}

@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: const Text('Archives'),
      actions: [
        IconButton(
          icon: const Icon(Icons.filter_list),
          tooltip: 'Filtres',
          onPressed:
              _hasConnection ? _showFilterDialog : null,
        ),
```

```
 ],
),
drawer: const AppDrawer(),
body: _isLoading
    ? const Center(child: CircularProgressIndicator())
    : _errorMessage != null
        ? Center(
            child: Padding(
              padding: const EdgeInsets.all(32),
              child: Column(
                mainAxisAlignment:
                    MainAxisAlignment.center,
                children: [
                  const Icon(Icons.cloud_off,
                      size: 80,
                      color: AppTheme.textSecondary),
                  const SizedBox(height: 16),
                  Text(
                    _errorMessage!,
                    textAlign: TextAlign.center,
                    style: const TextStyle(
                      fontSize: 16,
                      color: AppTheme.textSecondary,
                    ),
                  ),
                  const SizedBox(height: 24),
                  ElevatedButton.icon(
                    onPressed: _checkConnectionAndLoad,
                    icon: const Icon(Icons.refresh),
                    label: const Text('Réessayer'),
                  ),
                ],
              ),
            ),
          )
        : _archivedTasks.isEmpty
            ? Center(
                child: Column(
                  mainAxisAlignment:
                      MainAxisAlignment.center,
                  children: [
                    Icon(Icons.archive,
                        size: 80,
                        color: AppTheme.textSecondary
                            .withOpacity(0.3)),
                    const SizedBox(height: 16),
                    const Text(
                      'Aucune tâche archivée',
                      style: TextStyle(
                        fontSize: 18,
                        color: AppTheme.textSecondary,
                      ),
                    ),
                  ],
                ),
              )
            : RefreshIndicator(
                onRefresh: _loadArchivedTasks,
                child: Column(
                  children: [
                    Padding(
                      padding:
                          const EdgeInsets.all(12),
                      child: Text(
                        '${_archivedTasks.length} tâche(s) archivée(s)',
                        style: const TextStyle(
                          color:
                              AppTheme.textSecondary,
                          fontWeight: FontWeight.w600,
                        ),
                      ),
                    ),
                    Expanded(
                      child: ListView.builder(
                        padding:
                            const EdgeInsets.only(
                                bottom: 20),
                        itemCount:
                            _archivedTasks.length,
                        itemBuilder:
                            (context, index) {
                          final task =
                              _archivedTasks[index];
                          return TaskCard(
                            task: task,
                            onTap: () {
                              Navigator.push(
                                context,
                                MaterialPageRoute(
                                  builder: (_) =>
                                      TaskDetailScreen(
                                        task:
                                            task),
                                ),
                              );
                            },
                            onLongPress: () {
                              if (_auth.isAdmin) {
                                _unarchiveTask(
                                    task);
                              }
```

```
                        },
                      );
                    },
                  ),
                ),
              ],
            ),
          ),
        ),
      );
    }
}
```

dart file:<calendar_screen.dart>

```
// lib/screens/calendar_screen.dart
// ============================================
// ÉCRAN CALENDRIER DES TÂCHES PLANIFIÉES
// ============================================

import 'package:flutter/material.dart';
import 'package:intl/intl.dart';
import 'package:table_calendar/table_calendar.dart';
import '../models/task_model.dart';
import '../services/local_db_service.dart';
import '../utils/theme.dart';
import '../widgets/app_drawer.dart';
import '../widgets/task_card.dart';
import 'task_detail_screen.dart';
import 'task_form_screen.dart';

class CalendarScreen extends StatefulWidget {
  const CalendarScreen({super.key});

  @override
  State<CalendarScreen> createState() => _CalendarScreenState();
}

class _CalendarScreenState extends State<CalendarScreen> {
  final LocalDbService _localDb = LocalDbService();
  CalendarFormat _calendarFormat = CalendarFormat.month;
  DateTime _focusedDay = DateTime.now();
  DateTime _selectedDay = DateTime.now();
  Map<DateTime, List<TaskModel>> _tasksByDate = {};
  List<TaskModel> _selectedDayTasks = [];

  @override
  void initState() {
    super.initState();
    _loadPlannedTasks();
  }

  Future<void> _loadPlannedTasks() async {
    final tasks = await _localDb.getAllPlannedTasks();
    final Map<DateTime, List<TaskModel>> grouped = {};

    for (var task in tasks) {
      if (task.plannedDate != null) {
        final date = DateTime(
          task.plannedDate!.year,
          task.plannedDate!.month,
          task.plannedDate!.day,
        );
        grouped.putIfAbsent(date, () => []);
        grouped[date]!.add(task);
      }
    }

    if (mounted) {
      setState(() {
        _tasksByDate = grouped;
        _selectedDayTasks = _getTasksForDay(_selectedDay);
      });
    }
  }

  List<TaskModel> _getTasksForDay(DateTime day) {
    final normalizedDay = DateTime(day.year, day.month, day.day);
    return _tasksByDate[normalizedDay] ?? [];
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('Calendrier'),
      ),
      drawer: const AppDrawer(),
      floatingActionButton: FloatingActionButton(
        onPressed: () {
          Navigator.push(
            context,
            MaterialPageRoute(builder: (_) => const TaskFormScreen()),
          ).then((_) => _loadPlannedTasks());
        },
        child: const Icon(Icons.add),
      ),
      body: Column(
        children: [
          // Calendrier
          Card(
```

```
  margin: const EdgeInsets.all(8),
  child: TableCalendar<TaskModel>(
    firstDay: DateTime(2020),
    lastDay: DateTime(2030),
    focusedDay: _focusedDay,
    calendarFormat: _calendarFormat,
    locale: 'fr_FR',
    startingDayOfWeek: StartingDayOfWeek.monday,
    selectedDayPredicate: (day) => isSameDay(_selectedDay, day),
    eventLoader: _getTasksForDay,
    onDaySelected: (selectedDay, focusedDay) {
      setState(() {
        _selectedDay = selectedDay;
        _focusedDay = focusedDay;
        _selectedDayTasks = _getTasksForDay(selectedDay);
      });
    },
    onFormatChanged: (format) {
      setState(() => _calendarFormat = format);
    },
    onPageChanged: (focusedDay) {
      _focusedDay = focusedDay;
    },
    calendarStyle: CalendarStyle(
      todayDecoration: BoxDecoration(
        color: AppTheme.primaryColor.withOpacity(0.3),
        shape: BoxShape.circle,
      ),
      selectedDecoration: const BoxDecoration(
        color: AppTheme.primaryColor,
        shape: BoxShape.circle,
      ),
      markerDecoration: const BoxDecoration(
        color: AppTheme.secondaryColor,
        shape: BoxShape.circle,
      ),
      markerSize: 6,
      markersMaxCount: 3,
    ),
    headerStyle: const HeaderStyle(
      formatButtonVisible: true,
      titleCentered: true,
      formatButtonShowsNext: false,
    ),
  ),
),

// Date sélectionnée
Padding(
  padding: const EdgeInsets.symmetric(horizontal: 16, vertical: 8),
  child: Row(
    mainAxisAlignment: MainAxisAlignment.spaceBetween,
    children: [
      Text(
        DateFormat('EEEE d MMMM yyyy', 'fr_FR').format(_selectedDay),
        style: const TextStyle(
          fontSize: 16,
          fontWeight: FontWeight.w600,
          color: AppTheme.primaryColor,
        ),
      ),
      Text(
        '${_selectedDayTasks.length} tâche(s)',
        style: const TextStyle(color: AppTheme.textSecondary),
      ),
    ],
  ),
),

// Liste des tâches du jour sélectionné
Expanded(
  child: _selectedDayTasks.isEmpty
      ? Center(
          child: Column(
            mainAxisAlignment: MainAxisAlignment.center,
            children: [
              Icon(Icons.event_available,
                  size: 60,
                  color: AppTheme.textSecondary.withOpacity(0.3)),
              const SizedBox(height: 12),
              const Text(
                'Aucune tâche planifiée ce jour',
                style: TextStyle(color: AppTheme.textSecondary),
              ),
            ],
          ),
        )
      : ListView.builder(
          itemCount: _selectedDayTasks.length,
          itemBuilder: (context, index) {
            final task = _selectedDayTasks[index];
            return TaskCard(
              task: task,
              onTap: () {
                Navigator.push(
                  context,
                  MaterialPageRoute(
                    builder: (_) => TaskDetailScreen(task: task),
                  ),
                ).then((_) => _loadPlannedTasks());
              },
```

```
                  );
                },
              ),
          ),
        ],
      ),
    );
  }
}
```

dart file:<home_screen.dart>

```
// lib/screens/home_screen.dart
// ============================================
// ÉCRAN D'ACCUEIL
// ============================================

import 'package:flutter/material.dart';
import '../services/auth_service.dart';
import '../services/sync_service.dart';
import '../services/local_db_service.dart';
import '../services/notification_service.dart';
import '../utils/theme.dart';
import '../widgets/app_drawer.dart';
import 'task_list_screen.dart';
import 'task_form_screen.dart';
import 'calendar_screen.dart';
import 'archive_screen.dart';
import 'report_screen.dart';
import 'user_management_screen.dart';

class HomeScreen extends StatefulWidget {
  const HomeScreen({super.key});

  @override
  State<HomeScreen> createState() => _HomeScreenState();
}

class _HomeScreenState extends State<HomeScreen> {
  final AuthService _auth = AuthService();
  final SyncService _syncService = SyncService();
  final LocalDbService _localDb = LocalDbService();
  bool _isSyncing = false;
  int _pendingTaskCount = 0;
  int _doneTaskCount = 0;
  int _totalTaskCount = 0;

  @override
  void initState() {
    super.initState();
    _loadStats();
    _autoSync();
  }

  Future<void> _loadStats() async {
    final pendingTasks = await _localDb.getPendingTasks();
    final doneTasks = await _localDb.getDoneTasks();
    if (mounted) {
      setState(() {
        _pendingTaskCount = pendingTasks.length;
        _doneTaskCount = doneTasks.length;
        _totalTaskCount = pendingTasks.length + doneTasks.length;
      });
    }
  }

  Future<void> _autoSync() async {
    if (await _syncService.hasConnection()) {
      setState(() => _isSyncing = true);
      final result = await _syncService.syncAll();
      if (_auth.currentUser != null) {
        await NotificationService()
            .checkServerNotifications(_auth.currentUser!.id);
      }
      await _loadStats();
      if (mounted) {
        setState(() => _isSyncing = false);
        if (result.success && result.count > 0) {
          ScaffoldMessenger.of(context).showSnackBar(
            SnackBar(
              content: Text('□ ${result.message}'),
              backgroundColor: AppTheme.successColor,
            ),
          );
        }
      }
    }
  }

  Future<void> _manualSync() async {
    setState(() => _isSyncing = true);
    final result = await _syncService.syncAll();
    await _loadStats();
    if (mounted) {
      setState(() => _isSyncing = false);
      ScaffoldMessenger.of(context).showSnackBar(
        SnackBar(
          content: Text(
              result.success ? '□ ${result.message}' : '□ ${result.message}'),
          backgroundColor:
```

```
          result.success ? AppTheme.successColor : AppTheme.errorColor,
      ),
    );
  }
}

@override
Widget build(BuildContext context) {
  final isAdmin = _auth.isAdmin;

  return Scaffold(
    appBar: AppBar(
      title: const Text('Accueil'),
      actions: [
        if (_isSyncing)
          const Padding(
            padding: EdgeInsets.all(16),
            child: SizedBox(
              width: 20,
              height: 20,
              child: CircularProgressIndicator(
                color: Colors.white,
                strokeWidth: 2,
              ),
            ),
          )
        else
          IconButton(
            icon: const Icon(Icons.sync),
            tooltip: 'Synchroniser',
            onPressed: _manualSync,
          ),
      ],
    ),
    drawer: const AppDrawer(),
    body: RefreshIndicator(
      onRefresh: _manualSync,
      child: SingleChildScrollView(
        physics: const AlwaysScrollableScrollPhysics(),
        padding: const EdgeInsets.all(16),
        child: Column(
          crossAxisAlignment: CrossAxisAlignment.start,
          children: [
            // Bienvenue
            Card(
              child: Padding(
                padding: const EdgeInsets.all(20),
                child: Row(
                  children: [
                    CircleAvatar(
                      radius: 30,
                      backgroundColor:
                          AppTheme.primaryColor.withOpacity(0.1),
                      child: const Icon(Icons.person,
                          size: 30, color: AppTheme.primaryColor),
                    ),
                    const SizedBox(width: 16),
                    Expanded(
                      child: Column(
                        crossAxisAlignment: CrossAxisAlignment.start,
                        children: [
                          Text(
                            'Bonjour ${_auth.currentUser?.prenom ?? ""} !',
                            style: const TextStyle(
                              fontSize: 22,
                              fontWeight: FontWeight.bold,
                            ),
                          ),
                          Text(
                            isAdmin
                                ? '□ Administrateur'
                                : '□ Exécutant',
                            style: const TextStyle(
                              fontSize: 14,
                              color: AppTheme.textSecondary,
                            ),
                          ),
                        ],
                      ),
                    ),
                  ],
                ),
              ),
            ),
            const SizedBox(height: 16),

            // Statistiques
            Row(
              children: [
                // Carte tâches en cours
                Expanded(
                  flex: 3,
                  child: Card(
                    color: AppTheme.warningColor,
                    child: Padding(
                      padding: const EdgeInsets.all(14),
                      child: Column(
                        children: [
                          const Icon(Icons.pending_actions, color: Colors.white, size: 30),
                          const SizedBox(height: 6),
                          Text(
```

```
                '$_pendingTaskCount',
                style: const TextStyle(
                  fontSize: 26,
                  fontWeight: FontWeight.bold,
                  color: Colors.white,
                ),
              ),
              const Text(
                'En cours',
                style: TextStyle(color: Colors.white70, fontSize: 12),
              ),
            ],
          ),
        ),
      ),
    ),
    // Carte tâches terminées (plus large)
    Expanded(
      flex: 4,
      child: Card(
        color: AppTheme.successColor,
        child: Padding(
          padding: const EdgeInsets.all(14),
          child: Column(
            children: [
              const Icon(Icons.check_circle, color: Colors.white, size: 30),
              const SizedBox(height: 6),
              Text(
                '$_doneTaskCount',
                style: const TextStyle(
                  fontSize: 26,
                  fontWeight: FontWeight.bold,
                  color: Colors.white,
                ),
              ),
              const Text(
                'Terminées',
                style: TextStyle(color: Colors.white70, fontSize: 12),
              ),
            ],
          ),
        ),
      ),
    ),
    // Carte total
    Expanded(
      flex: 3,
      child: Card(
        color: AppTheme.primaryColor,
        child: Padding(
          padding: const EdgeInsets.all(14),
          child: Column(
            children: [
              const Icon(Icons.assignment, color: Colors.white, size: 30),
              const SizedBox(height: 6),
              Text(
                '$_totalTaskCount',
                style: const TextStyle(
                  fontSize: 26,
                  fontWeight: FontWeight.bold,
                  color: Colors.white,
                ),
              ),
              const Text(
                'Total',
                style: TextStyle(color: Colors.white70, fontSize: 12),
              ),
            ],
          ),
        ),
      ),
    ),
  ],
),
const SizedBox(height: 24),

// Menu rapide
const Text(
  'Accès rapide',
  style: TextStyle(fontSize: 20, fontWeight: FontWeight.bold),
),
const SizedBox(height: 12),

GridView.count(
  shrinkWrap: true,
  physics: const NeverScrollableScrollPhysics(),
  crossAxisCount: 2,
  mainAxisSpacing: 12,
  crossAxisSpacing: 12,
  childAspectRatio: 1.3,
  children: [
    _buildQuickAction(
      icon: Icons.add_task,
      label: 'Nouvelle\ntâche',
      color: AppTheme.secondaryColor,
      onTap: () {
        Navigator.push(
          context,
          MaterialPageRoute(
              builder: (_) => const TaskFormScreen()),
        ).then((_) => _loadStats());
```

```
                  },
                ),
                _buildQuickAction(
                  icon: Icons.list_alt,
                  label: 'Liste des\ntâches',
                  color: AppTheme.primaryColor,
                  onTap: () {
                    Navigator.push(
                      context,
                      MaterialPageRoute(
                          builder: (_) => const TaskListScreen()),
                    );
                  },
                ),
                _buildQuickAction(
                  icon: Icons.calendar_month,
                  label: 'Calendrier',
                  color: AppTheme.warningColor,
                  onTap: () {
                    Navigator.push(
                      context,
                      MaterialPageRoute(
                          builder: (_) => const CalendarScreen()),
                    );
                  },
                ),
                _buildQuickAction(
                  icon: Icons.assessment,
                  label: 'Rapports',
                  color: Colors.deepPurple,
                  onTap: () {
                    Navigator.push(
                      context,
                      MaterialPageRoute(
                          builder: (_) => const ReportScreen()),
                    );
                  },
                ),
                if (isAdmin)
                  _buildQuickAction(
                    icon: Icons.archive,
                    label: 'Archives',
                    color: AppTheme.archiveColor,
                    onTap: () {
                      Navigator.push(
                        context,
                        MaterialPageRoute(
                            builder: (_) => const ArchiveScreen()),
                      );
                    },
                  ),
                if (isAdmin)
                  _buildQuickAction(
                    icon: Icons.people,
                    label: 'Utilisateurs',
                    color: Colors.teal,
                    onTap: () {
                      Navigator.push(
                        context,
                        MaterialPageRoute(
                            builder: (_) =>
                                const UserManagementScreen()),
                      );
                    },
                  ),
              ],
            ),
          ],
        ),
      ),
    ),
  );
}

Widget _buildQuickAction({
  required IconData icon,
  required String label,
  required Color color,
  required VoidCallback onTap,
}) {
  return Card(
    elevation: 3,
    child: InkWell(
      borderRadius: BorderRadius.circular(12),
      onTap: onTap,
      child: Padding(
        padding: const EdgeInsets.all(16),
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: [
            Icon(icon, size: 36, color: color),
            const SizedBox(height: 8),
            Text(
              label,
              textAlign: TextAlign.center,
              style: TextStyle(
                fontSize: 14,
                fontWeight: FontWeight.w600,
                color: color,
              ),
            ),
```

```
            ],
          ),
        ),
      ),
    );
  }
}


***dart file:<login_screen.dart>***

// lib/screens/login_screen.dart
// ============================================
// ÉCRAN DE CONNEXION
// ============================================

import 'package:flutter/material.dart';
import '../services/auth_service.dart';
import '../services/sync_service.dart';
import '../services/notification_service.dart';
import '../utils/theme.dart';
import '../widgets/app_text_field.dart';
import 'home_screen.dart';

class LoginScreen extends StatefulWidget {
  const LoginScreen({super.key});

  @override
  State<LoginScreen> createState() => _LoginScreenState();
}

class _LoginScreenState extends State<LoginScreen> {
  final _formKey = GlobalKey<FormState>();
  final _identifiantController = TextEditingController();
  final _passwordController = TextEditingController();
  bool _isLoading = false;
  bool _obscurePassword = true;
  String? _errorMessage;

  @override
  void dispose() {
    _identifiantController.dispose();
    _passwordController.dispose();
    super.dispose();
  }

  Future<void> _login() async {
    if (!_formKey.currentState!.validate()) return;

    setState(() {
      _isLoading = true;
      _errorMessage = null;
    });

    try {
      final user = await AuthService().login(
        _identifiantController.text.trim(),
        _passwordController.text,
      );

      if (user != null) {
        // Synchroniser au démarrage si connexion disponible
        final syncService = SyncService();
        if (await syncService.hasConnection()) {
          await syncService.syncAll();
          // Vérifier les notifications
          await NotificationService().checkServerNotifications(user.id);
        }

        if (mounted) {
          Navigator.pushReplacement(
            context,
            MaterialPageRoute(builder: (_) => const HomeScreen()),
          );
        }
      } else {
        setState(() {
          _errorMessage = 'Identifiant ou mot de passe incorrect';
        });
      }
    } catch (e) {
      setState(() {
        _errorMessage = 'Erreur de connexion. Vérifiez votre réseau.';
      });
    } finally {
      if (mounted) {
        setState(() {
          _isLoading = false;
        });
      }
    }
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      backgroundColor: AppTheme.backgroundColor,
      body: SafeArea(
        child: Center(
          child: SingleChildScrollView(
            padding: const EdgeInsets.all(24),
```

```
child: Form(
  key: _formKey,
  child: Column(
    mainAxisAlignment: MainAxisAlignment.center,
    children: [
      // Logo / Icône
      Container(
        padding: const EdgeInsets.all(20),
        decoration: BoxDecoration(
          color: AppTheme.primaryColor.withValues(alpha: 25),
          shape: BoxShape.circle,
        ),
        child: const Icon(
          Icons.apartment,
          size: 80,
          color: AppTheme.primaryColor,
        ),
      ),
      const SizedBox(height: 24),

      // Titre
      const Text(
        'Entretien Immeuble',
        style: TextStyle(
          fontSize: 28,
          fontWeight: FontWeight.bold,
          color: AppTheme.primaryColor,
        ),
      ),
      const SizedBox(height: 8),
      const Text(
        'Connectez-vous pour continuer',
        style: TextStyle(
          fontSize: 16,
          color: AppTheme.textSecondary,
        ),
      ),
      const SizedBox(height: 40),

      // Message d'erreur
      if (_errorMessage != null) ...[
        Container(
          padding: const EdgeInsets.all(12),
          decoration: BoxDecoration(
            color: AppTheme.errorColor.withValues(alpha: 25),
            borderRadius: BorderRadius.circular(10),
            border: Border.all(
                color: AppTheme.errorColor.withValues(alpha: 77)),
          ),
          child: Row(
            children: [
              const Icon(Icons.error_outline,
                  color: AppTheme.errorColor),
              const SizedBox(width: 8),
              Expanded(
                child: Text(
                  _errorMessage!,
                  style:
                      const TextStyle(color: AppTheme.errorColor),
                ),
              ),
            ],
          ),
        ),
        const SizedBox(height: 16),
      ],

      // Champ identifiant
      AppTextField(
        controller: _identifiantController,
        labelText: 'Identifiant',
        prefixIcon: const Icon(Icons.person),
        validator: (value) {
          if (value == null || value.trim().isEmpty) {
            return 'Veuillez entrer votre identifiant';
          }
          return null;
        },
      ),
      const SizedBox(height: 16),

      // Champ mot de passe
      AppTextField(
        controller: _passwordController,
        obscureText: _obscurePassword,
        labelText: 'Mot de passe',
        prefixIcon: const Icon(Icons.lock),
        suffixIcon: IconButton(
          icon: Icon(_obscurePassword
              ? Icons.visibility_off
              : Icons.visibility),
          onPressed: () {
            setState(() {
              _obscurePassword = !_obscurePassword;
            });
          },
        ),
        validator: (value) {
          if (value == null || value.isEmpty) {
            return 'Veuillez entrer votre mot de passe';
          }
```

```
                  return null;
                },
                onFieldSubmitted: (_) => _login(),
              ),
              const SizedBox(height: 32),

              // Bouton connexion
              SizedBox(
                width: double.infinity,
                height: 50,
                child: ElevatedButton(
                  onPressed: _isLoading ? null : _login,
                  child: _isLoading
                      ? const SizedBox(
                          width: 24,
                          height: 24,
                          child: CircularProgressIndicator(
                            color: Colors.white,
                            strokeWidth: 2,
                          ),
                        )
                      : const Text(
                          'Se connecter',
                          style: TextStyle(fontSize: 18),
                        ),
                ),
              ),
            ],
          ),
        ),
      ),
    ),
  );
 }
}


***dart file:<report_screen.dart>***

// lib/screens/report_screen.dart
// ================================================
// ÉCRAN GÉNÉRATION DE RAPPORTS
// ================================================
import 'package:flutter/material.dart';
import 'package:intl/intl.dart';
import 'package:pdf/pdf.dart';
import 'package:pdf/widgets.dart' as pw;
import 'package:printing/printing.dart';
import 'package:share_plus/share_plus.dart';
import 'dart:typed_data';
import 'package:path_provider/path_provider.dart';
import 'dart:io';
import '../models/task_model.dart';
import '../models/immeuble_model.dart';
import '../services/supabase_service.dart';
import '../services/local_db_service.dart';
import '../services/sync_service.dart';
import '../utils/theme.dart';
import '../widgets/app_drawer.dart';

class ReportScreen extends StatefulWidget {
  const ReportScreen({super.key});

  @override
  State<ReportScreen> createState() => _ReportScreenState();
}

class _ReportScreenState extends State<ReportScreen> {
  final SupabaseService _supabase = SupabaseService();
  final LocalDbService _localDb = LocalDbService();

  List<TaskModel> _reportTasks = [];
  bool _isLoading = false;
  bool _hasSearched = false;

  // Liste des immeubles
  List<ImmeubleModel> _immeubles = [];
  String? _selectedImmeuble;

  // Filtres
  final TextEditingController _etageFilter = TextEditingController();
  final TextEditingController _chambreFilter = TextEditingController();
  final TextEditingController _executantFilter = TextEditingController();
  DateTime? _dateFilter;
  String? _statusFilter;
  String _sortBy = 'created_at';
  bool _sortAscending = false;

  @override
  void initState() {
    super.initState();
    _loadImmeubles();
  }

  @override
  void dispose() {
    _etageFilter.dispose();
    _chambreFilter.dispose();
    _executantFilter.dispose();
    super.dispose();
```

```
}

Future<void> _loadImmeubles() async {
  final immeubles = await _localDb.getActiveImmeubles();
  if (mounted) {
    setState(() {
      _immeubles = immeubles;
    });
  }
}

Future<void> _generateReport() async {
  if (!await SyncService().hasConnection()) {
    if (mounted) {
      ScaffoldMessenger.of(context).showSnackBar(
        const SnackBar(
          content: Text(
              '□ Connexion internet requise pour les rapports'),
          backgroundColor: AppTheme.errorColor,
        ),
      );
    }
    return;
  }

  setState(() {
    _isLoading = true;
    _hasSearched = true;
  });

  try {
    final tasks = await _supabase.getTasksReport(
      immeuble: _selectedImmeuble,
      etage: _etageFilter.text.trim().isNotEmpty
          ? _etageFilter.text.trim()
          : null,
      chambre: _chambreFilter.text.trim().isNotEmpty
          ? _chambreFilter.text.trim()
          : null,
      doneBy: _executantFilter.text.trim().isNotEmpty
          ? _executantFilter.text.trim()
          : null,
      doneDate: _dateFilter,
      status: _statusFilter,
      orderBy: _sortBy,
      ascending: _sortAscending,
    );

    if (mounted) {
      setState(() {
        _reportTasks = tasks;
        _isLoading = false;
      });
    }
  } catch (e) {
    if (mounted) {
      setState(() => _isLoading = false);
      ScaffoldMessenger.of(context).showSnackBar(
        SnackBar(
          content: Text('□ Erreur: $e'),
          backgroundColor: AppTheme.errorColor,
        ),
      );
    }
  }
}

Future<Uint8List> _buildPdf() async {
  final pdf = pw.Document();
  final dateFormat = DateFormat('dd/MM/yyyy');

  pdf.addPage(
    pw.MultiPage(
      pageFormat: PdfPageFormat.a4,
      margin: const pw.EdgeInsets.all(32),
      header: (context) => pw.Column(
        crossAxisAlignment: pw.CrossAxisAlignment.start,
        children: [
          pw.Text(
            'Rapport d\'entretien d\'immeuble',
            style: pw.TextStyle(
                fontSize: 20,
                fontWeight: pw.FontWeight.bold),
          ),
          pw.SizedBox(height: 4),
          pw.Text(
            'Généré le ${dateFormat.format(DateTime.now())} — ${_reportTasks.length} tâche(s)',
            style: const pw.TextStyle(
                fontSize: 10, color: PdfColors.grey700),
          ),
          pw.Divider(),
          pw.SizedBox(height: 8),
        ],
      ),
      build: (context) => [
        pw.Table.fromTextArray(
          headerStyle: pw.TextStyle(
            fontWeight: pw.FontWeight.bold,
            fontSize: 9,
          ),
          cellStyle: const pw.TextStyle(fontSize: 8),
```

```
          headerDecoration: const pw.BoxDecoration(
            color: PdfColors.grey300,
          ),
          cellAlignments: {
            0: pw.Alignment.centerLeft,
            1: pw.Alignment.centerLeft,
            2: pw.Alignment.center,
            3: pw.Alignment.center,
            4: pw.Alignment.centerLeft,
            5: pw.Alignment.center,
            6: pw.Alignment.center,
          },
          headers: [
            'Immeuble',
            'Description',
            'Étage',
            'Ch.',
            'Exécutant',
            'Statut',
            'Date',
          ],
          data: _reportTasks.map((task) {
            return [
              task.immeuble,
              task.description.length > 40
                  ? '${task.description.substring(0, 40)}...'
                  : task.description,
              task.etage,
              task.chambre,
              task.doneBy,
              task.statusText,
              task.doneDate != null
                  ? dateFormat.format(task.doneDate!)
                  : task.plannedDate != null
                      ? dateFormat
                          .format(task.plannedDate!)
                      : '',
            ];
          }).toList(),
        ),
      ],
    ),
  );

  return pdf.save();
}

Future<void> _sharePdf() async {
  if (_reportTasks.isEmpty) {
    if (mounted) {
      ScaffoldMessenger.of(context).showSnackBar(
        const SnackBar(
          content: Text('□ Aucune tâche dans le rapport'),
          backgroundColor: AppTheme.errorColor,
        ),
      );
    }
    return;
  }

  try {
    if (mounted) {
      ScaffoldMessenger.of(context).showSnackBar(
        const SnackBar(
          content: Row(
            children: [
              SizedBox(
                width: 20,
                height: 20,
                child: CircularProgressIndicator(
                  color: Colors.white,
                  strokeWidth: 2,
                ),
              ),
              SizedBox(width: 12),
              Text('Génération du PDF...'),
            ],
          ),
          duration: Duration(seconds: 2),
        ),
      );
    }

    final pdfBytes = await _buildPdf();
    final dir = await getApplicationCacheDirectory();
    final fileName =
        'rapport_entretien_${DateFormat('yyyyMMdd_HHmmss').format(DateTime.now())}.pdf';
    final file = File('${dir.path}/$fileName');
    await file.writeAsBytes(pdfBytes);

    if (await file.exists() && await file.length() > 0) {
      await Share.shareXFiles(
        [XFile(file.path, mimeType: 'application/pdf')],
        subject:
            'Rapport d\'entretien - ${DateFormat('dd/MM/yyyy').format(DateTime.now())}',
        text:
            'Veuillez trouver ci-joint le rapport d\'entretien d\'immeuble.',
      );
    } else {
      throw Exception('Le fichier PDF n\'a pas pu être créé');
    }
```

```
  } catch (e) {
    if (mounted) {
      ScaffoldMessenger.of(context).showSnackBar(
        SnackBar(
          content: Text('□ Erreur de partage: $e'),
          backgroundColor: AppTheme.errorColor,
        ),
      );
    }
  }
}

Future<void> _printPdf() async {
  if (_reportTasks.isEmpty) {
    if (mounted) {
      ScaffoldMessenger.of(context).showSnackBar(
        const SnackBar(
          content: Text('□ Aucune tâche dans le rapport'),
          backgroundColor: AppTheme.errorColor,
        ),
      );
    }
    return;
  }

  try {
    final pdfBytes = await _buildPdf();
    await Printing.layoutPdf(onLayout: (_) => pdfBytes);
  } catch (e) {
    if (mounted) {
      ScaffoldMessenger.of(context).showSnackBar(
        SnackBar(
          content: Text('□ Erreur d\'impression: $e'),
          backgroundColor: AppTheme.errorColor,
        ),
      );
    }
  }
}

@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: const Text('Rapports'),
      actions: [
        if (_reportTasks.isNotEmpty) ...[
          IconButton(
            icon: const Icon(Icons.share),
            tooltip: 'Partager / Email',
            onPressed: _sharePdf,
          ),
          IconButton(
            icon: const Icon(Icons.print),
            tooltip: 'Imprimer / PDF',
            onPressed: _printPdf,
          ),
        ],
      ],
    ),
    drawer: const AppDrawer(),
    body: SingleChildScrollView(
      padding: const EdgeInsets.all(16),
      child: Column(
        crossAxisAlignment: CrossAxisAlignment.start,
        children: [
          const Text(
            'Critères du rapport',
            style: TextStyle(
                fontSize: 20, fontWeight: FontWeight.bold),
          ),
          const SizedBox(height: 16),

          // ==========================================
          // IMMEUBLE — Liste déroulante
          // ==========================================
          DropdownButtonFormField<String>(
            value: _selectedImmeuble,
            decoration: const InputDecoration(
              labelText: 'Immeuble',
              prefixIcon: Icon(Icons.apartment),
            ),
            isExpanded: true,
            hint: const Text('Tous les immeubles'),
            items: [
              const DropdownMenuItem<String>(
                value: null,
                child: Text('Tous les immeubles'),
              ),
              ..._immeubles.map((immeuble) {
                return DropdownMenuItem<String>(
                  value: immeuble.nom,
                  child: Text(immeuble.nom),
                );
              }),
            ],
            onChanged: (value) {
              setState(() {
                _selectedImmeuble = value;
              });
            },
```

```
),
const SizedBox(height: 12),

// Étage et chambre
Row(
  children: [
    Expanded(
      child: TextField(
        controller: _etageFilter,
        decoration: const InputDecoration(
          labelText: 'Étage',
          prefixIcon: Icon(Icons.layers),
        ),
      ),
    ),
    const SizedBox(width: 12),
    Expanded(
      child: TextField(
        controller: _chambreFilter,
        decoration: const InputDecoration(
          labelText: 'Chambre',
          prefixIcon:
              Icon(Icons.door_front_door),
        ),
      ),
    ),
  ],
),
const SizedBox(height: 12),

// Exécutant
TextField(
  controller: _executantFilter,
  decoration: const InputDecoration(
    labelText: 'Exécutant',
    prefixIcon: Icon(Icons.person),
  ),
),
const SizedBox(height: 12),

// Date
Card(
  child: ListTile(
    leading: const Icon(Icons.calendar_today),
    title: Text(_dateFilter != null
        ? DateFormat('dd/MM/yyyy')
            .format(_dateFilter!)
        : 'Date d\'exécution'),
    trailing: Row(
      mainAxisSize: MainAxisSize.min,
      children: [
        IconButton(
          icon: const Icon(Icons.edit_calendar),
          onPressed: () async {
            final picked =
                await showDatePicker(
              context: context,
              initialDate:
                  _dateFilter ?? DateTime.now(),
              firstDate: DateTime(2020),
              lastDate: DateTime(2030),
            );
            if (picked != null) {
              setState(
                  () => _dateFilter = picked);
            }
          },
        ),
        if (_dateFilter != null)
          IconButton(
            icon: const Icon(Icons.clear,
                color: AppTheme.errorColor),
            onPressed: () {
              setState(
                  () => _dateFilter = null);
            },
          ),
      ],
    ),
  ),
),
const SizedBox(height: 12),

// Statut
const Text('Statut :',
    style:
        TextStyle(fontWeight: FontWeight.w600)),
const SizedBox(height: 8),
Wrap(
  spacing: 8,
  children: [
    ChoiceChip(
      label: const Text('Tous'),
      selected: _statusFilter == null,
      onSelected: (_) => setState(
          () => _statusFilter = null),
    ),
    ChoiceChip(
      label: const Text('En cours'),
      selected: _statusFilter == 'pending',
      onSelected: (_) => setState(
```

```
          () => _statusFilter = 'pending'),
    ),
    ChoiceChip(
      label: const Text('Terminées'),
      selected: _statusFilter == 'done',
      onSelected: (_) => setState(
          () => _statusFilter = 'done'),
    ),
    ChoiceChip(
      label: const Text('Archivées'),
      selected: _statusFilter == 'archived',
      onSelected: (_) => setState(
          () => _statusFilter = 'archived'),
    ),
  ],
),
const SizedBox(height: 16),

// Tri
const Text('Trier par :',
    style:
        TextStyle(fontWeight: FontWeight.w600)),
const SizedBox(height: 8),
Wrap(
  spacing: 8,
  children: [
    ChoiceChip(
      label: const Text('Date création'),
      selected: _sortBy == 'created_at',
      onSelected: (_) => setState(
          () => _sortBy = 'created_at'),
    ),
    ChoiceChip(
      label: const Text('Immeuble'),
      selected: _sortBy == 'immeuble',
      onSelected: (_) => setState(
          () => _sortBy = 'immeuble'),
    ),
    ChoiceChip(
      label: const Text('Date exéc.'),
      selected: _sortBy == 'done_date',
      onSelected: (_) => setState(
          () => _sortBy = 'done_date'),
    ),
  ],
),
const SizedBox(height: 24),

// Bouton générer
SizedBox(
  width: double.infinity,
  height: 50,
  child: ElevatedButton.icon(
    onPressed:
        _isLoading ? null : _generateReport,
    icon: _isLoading
        ? const SizedBox(
            width: 20,
            height: 20,
            child: CircularProgressIndicator(
              color: Colors.white,
              strokeWidth: 2,
            ),
          )
        : const Icon(Icons.assessment),
    label: Text(_isLoading
        ? 'Chargement...'
        : 'Générer le rapport'),
  ),
),
const SizedBox(height: 24),

// Résultats
if (_hasSearched) ...[
  Row(
    mainAxisAlignment:
        MainAxisAlignment.spaceBetween,
    children: [
      Text(
        '${_reportTasks.length} résultat(s)',
        style: const TextStyle(
          fontSize: 18,
          fontWeight: FontWeight.bold,
        ),
      ),
      if (_reportTasks.isNotEmpty)
        Row(
          children: [
            IconButton(
              icon: const Icon(Icons.share,
                  color:
                      AppTheme.primaryColor),
              tooltip: 'Partager par email',
              onPressed: _sharePdf,
            ),
            IconButton(
              icon: const Icon(Icons.print,
                  color:
                      AppTheme.primaryColor),
              tooltip: 'Imprimer',
              onPressed: _printPdf,
```

```
                    ),
                  ],
                ),
              ],
            ),
            const Divider(),

            // Liste des résultats
            ..._reportTasks.map((task) => Card(
                  margin:
                      const EdgeInsets.only(bottom: 8),
                  child: ListTile(
                    leading: Icon(
                      task.done
                          ? Icons.check_circle
                          : task.archived
                              ? Icons.archive
                              : Icons.pending,
                      color: task.done
                          ? AppTheme.successColor
                          : task.archived
                              ? AppTheme.archiveColor
                              : AppTheme.warningColor,
                    ),
                    title: Text(
                      task.immeuble,
                      style: const TextStyle(
                          fontWeight: FontWeight.w600),
                    ),
                    subtitle: Column(
                      crossAxisAlignment:
                          CrossAxisAlignment.start,
                      children: [
                        Text(task.description,
                            maxLines: 2,
                            overflow:
                                TextOverflow.ellipsis),
                        if (task.etage.isNotEmpty ||
                            task.chambre.isNotEmpty)
                          Text(
                            '${task.etage.isNotEmpty ? "Ét. ${task.etage}" : ""}${task.chambre.isNotEmpty ? " Ch. ${task.chambre}" : ""}',
                            style: const TextStyle(
                                fontSize: 12,
                                color: AppTheme
                                    .textSecondary),
                          ),
                        Text(
                          '${task.statusText}${task.doneBy.isNotEmpty ? " — ${task.doneBy}" : ""}',
                          style: const TextStyle(
                              fontSize: 12,
                              color: AppTheme
                                  .textSecondary),
                        ),
                      ],
                    ),
                    isThreeLine: true,
                  ),
                )),
          ],
        ],
      ),
    ),
  );
 }
}


***dart file:<task_detail_screen.dart>***

// lib/screens/task_detail_screen.dart
// ============================================
// ÉCRAN DÉTAIL D'UNE TÂCHE
// ============================================

import 'package:flutter/material.dart';
import 'package:intl/intl.dart';
import '../models/task_model.dart';
import '../models/task_history_model.dart';
import '../services/local_db_service.dart';
import '../services/supabase_service.dart';
import '../services/sync_service.dart';
import '../utils/theme.dart';
import 'task_form_screen.dart';
import 'task_history_screen.dart';

class TaskDetailScreen extends StatefulWidget {
  final TaskModel task;
  final bool showHistory;

  const TaskDetailScreen({
    super.key,
    required this.task,
    this.showHistory = false,
  });

  @override
  State<TaskDetailScreen> createState() => _TaskDetailScreenState();
}

class _TaskDetailScreenState extends State<TaskDetailScreen> {
  late TaskModel _task;
```

```
@override
void initState() {
  super.initState();
  _task = widget.task;
  _refreshTask();
  if (widget.showHistory) {
    WidgetsBinding.instance.addPostFrameCallback((_) {
      Navigator.push(
        context,
        MaterialPageRoute(
          builder: (_) => TaskHistoryScreen(taskId: _task.id),
        ),
      );
    });
  }
}

Future<void> _refreshTask() async {
  final localTask = await LocalDbService().getTaskById(_task.id);
  if (localTask != null && mounted) {
    setState(() => _task = localTask);
  }
}

@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: Text('Tâche ${_task.displayNumber}'),
      actions: [
        IconButton(
          icon: const Icon(Icons.edit),
          tooltip: 'Modifier',
          onPressed: () {
            Navigator.push(
              context,
              MaterialPageRoute(
                builder: (_) => TaskFormScreen(task: _task),
              ),
            ).then((_) => _refreshTask());
          },
        ),
        IconButton(
          icon: const Icon(Icons.history),
          tooltip: 'Historique',
          onPressed: () {
            Navigator.push(
              context,
              MaterialPageRoute(
                builder: (_) => TaskHistoryScreen(taskId: _task.id),
              ),
            );
          },
        ),
      ],
    ),
    body: SingleChildScrollView(
      padding: const EdgeInsets.all(16),
      child: Column(
        crossAxisAlignment: CrossAxisAlignment.start,
        children: [
          // Statut
          Center(
            child: Container(
              padding:
                  const EdgeInsets.symmetric(horizontal: 20, vertical: 10),
              decoration: BoxDecoration(
                color: _task.done
                    ? AppTheme.successColor.withOpacity(0.1)
                    : AppTheme.warningColor.withOpacity(0.1),
                borderRadius: BorderRadius.circular(20),
                border: Border.all(
                  color: _task.done
                      ? AppTheme.successColor
                      : AppTheme.warningColor,
                ),
              ),
              child: Text(
                _task.statusText,
                style: TextStyle(
                  fontSize: 18,
                  fontWeight: FontWeight.bold,
                  color: _task.done
                      ? AppTheme.successColor
                      : AppTheme.warningColor,
                ),
              ),
            ),
          ),
          const SizedBox(height: 20),

          // Informations principales
          _buildInfoCard('Immeuble', _task.immeuble, Icons.apartment),
          if (_task.etage.isNotEmpty)
            _buildInfoCard('Étage', _task.etage, Icons.layers),
          if (_task.chambre.isNotEmpty)
            _buildInfoCard('Chambre', _task.chambre, Icons.door_front_door),
          _buildInfoCard('Description', _task.description, Icons.description),
          _buildInfoCard(
            'Date de création',
```

```
            DateFormat('dd/MM/yyyy à HH:mm').format(_task.createdAt),
            Icons.date_range,
          ),

          if (_task.plannedDate != null)
            _buildInfoCard(
              'Date planifiée',
              DateFormat('dd/MM/yyyy').format(_task.plannedDate!),
              Icons.calendar_today,
            ),

          if (_task.done) ...[
            const Divider(height: 32),
            const Text(
              'Exécution',
              style: TextStyle(fontSize: 18, fontWeight: FontWeight.bold),
            ),
            const SizedBox(height: 8),
            if (_task.doneDate != null)
              _buildInfoCard(
                'Date d\'exécution',
                DateFormat('dd/MM/yyyy').format(_task.doneDate!),
                Icons.event_available,
              ),
            if (_task.doneBy.isNotEmpty)
              _buildInfoCard('Exécutant', _task.doneBy, Icons.person),
          ],

          // Photo
          if (_task.photoUrl.isNotEmpty) ...[
            const Divider(height: 32),
            const Text(
              'Photo',
              style: TextStyle(fontSize: 18, fontWeight: FontWeight.bold),
            ),
            const SizedBox(height: 8),
            ClipRRect(
              borderRadius: BorderRadius.circular(12),
              child: Image.network(
                _task.photoUrl,
                width: double.infinity,
                fit: BoxFit.cover,
                errorBuilder: (_, __, ___) => Container(
                  height: 200,
                  color: Colors.grey[200],
                  child: const Center(
                    child: Icon(Icons.broken_image,
                        size: 60, color: AppTheme.textSecondary),
                  ),
                ),
              ),
            ),
          ],
        ],
      ),
    ),
  );
 }

 Widget _buildInfoCard(String label, String value, IconData icon) {
   return Card(
     margin: const EdgeInsets.only(bottom: 8),
     child: ListTile(
       leading: Icon(icon, color: AppTheme.primaryColor),
       title: Text(label,
           style: const TextStyle(fontSize: 12, color: AppTheme.textSecondary)),
       subtitle: Text(
         value,
         style: const TextStyle(fontSize: 16, fontWeight: FontWeight.w500),
       ),
     ),
   );
 }
}


***dart file:<task_form_screen.dart>***

// lib/screens/task_form_screen.dart
// ============================================
// ÉCRAN FORMULAIRE DE CRÉATION/MODIFICATION DE TÂCHE
// ============================================

import 'dart:io';
import 'package:flutter/material.dart';
import 'package:image_picker/image_picker.dart';
import 'package:intl/intl.dart';
import 'package:uuid/uuid.dart';
import '../models/task_model.dart';
import '../models/task_history_model.dart';
import '../models/immeuble_model.dart';
import '../services/local_db_service.dart';
import '../services/supabase_service.dart';
import '../services/auth_service.dart';
import '../services/sync_service.dart';
import '../services/notification_service.dart';
import '../utils/theme.dart';
import '../widgets/app_text_field.dart';

class TaskFormScreen extends StatefulWidget {
  final TaskModel? task;
```

```
  const TaskFormScreen({super.key, this.task});

  @override
  State<TaskFormScreen> createState() => _TaskFormScreenState();
}

class _TaskFormScreenState extends State<TaskFormScreen> {
  final _formKey = GlobalKey<FormState>();
  final LocalDbService _localDb = LocalDbService();
  final SupabaseService _supabase = SupabaseService();
  final AuthService _auth = AuthService();

  late TextEditingController _etageController;
  late TextEditingController _chambreController;
  late TextEditingController _descriptionController;
  late TextEditingController _doneByController;

  // Liste des immeubles
  List<ImmeubleModel> _immeubles = [];
  String? _selectedImmeuble;

  bool _done = false;
  DateTime? _doneDate;
  DateTime? _plannedDate;
  String? _photoLocalPath;
  String? _photoUrl;
  bool _isSaving = false;

  bool get _isEditing => widget.task != null;

  @override
  void initState() {
    super.initState();
    _etageController = TextEditingController(text: widget.task?.etage ?? '');
    _chambreController = TextEditingController(text: widget.task?.chambre ?? '');
    _descriptionController =
        TextEditingController(text: widget.task?.description ?? '');
    _doneByController = TextEditingController(text: widget.task?.doneBy ?? '');
    _selectedImmeuble =
        widget.task?.immeuble.isNotEmpty == true ? widget.task!.immeuble : null;
    _done = widget.task?.done ?? false;
    _doneDate = widget.task?.doneDate;
    _plannedDate = widget.task?.plannedDate;
    _photoUrl = widget.task?.photoUrl;
    _photoLocalPath = widget.task?.photoLocalPath;
    _loadImmeubles();
  }

  @override
  void dispose() {
    _etageController.dispose();
    _chambreController.dispose();
    _descriptionController.dispose();
    _doneByController.dispose();
    super.dispose();
  }

  Future<void> _loadImmeubles() async {
    final immeubles = await _localDb.getActiveImmeubles();
    if (mounted) {
      setState(() {
        _immeubles = immeubles;
        // Si l'immeuble de la tâche n'est pas dans la liste, l'ajouter
        if (_selectedImmeuble != null &&
            !_immeubles.any((i) => i.nom == _selectedImmeuble)) {
          _immeubles.add(ImmeubleModel(
            id: 'temp',
            nom: _selectedImmeuble!,
          ));
        }
      });
    }
  }

  void _showAddImmeubleDialog() {
    final controller = TextEditingController();
    showDialog(
      context: context,
      builder: (context) => AlertDialog(
        title: const Text('Ajouter un immeuble'),
        content: AppTextField(
          controller: controller,
          labelText: 'Nom de l\'immeuble',
          prefixIcon: const Icon(Icons.apartment),
        ),
        actions: [
          TextButton(
            onPressed: () => Navigator.pop(context),
            child: const Text('Annuler'),
          ),
          ElevatedButton(
            onPressed: () async {
              final nom = controller.text.trim();
              if (nom.isNotEmpty) {
                // Ajouter en local
                await _localDb.insertImmeubleIfNotExists(nom);

                // Ajouter sur Supabase si connecté
                if (await SyncService().hasConnection()) {
                  try {
```

```
                await _supabase.insertImmeubleIfNotExists(nom);
              } catch (e) {
                // Sera synchronisé plus tard
              }
            }

            // Recharger la liste
            await _loadImmeubles();

            if (mounted) {
              setState(() {
                _selectedImmeuble = nom;
              });
              Navigator.pop(context);
            }
          }
        },
        child: const Text('Ajouter'),
      ),
    ],
  ),
 );
}

Future<void> _pickImage() async {
 final picker = ImagePicker();
 FocusScope.of(context).unfocus();
 await Future.delayed(const Duration(milliseconds: 200));
 if (!mounted) return;

 final ImageSource? source = await showDialog<ImageSource>(
  context: context,
  builder: (BuildContext dialogContext) {
   return AlertDialog(
    title: const Text('Choisir une source'),
    content: Column(
     mainAxisSize: MainAxisSize.min,
     children: [
      ListTile(
       leading:
           const Icon(Icons.camera_alt, color: AppTheme.primaryColor),
       title: const Text('Prendre une photo'),
       onTap: () =>
           Navigator.of(dialogContext).pop(ImageSource.camera),
      ),
      ListTile(
       leading: const Icon(Icons.photo_library,
           color: AppTheme.secondaryColor),
       title: const Text('Galerie photos'),
       onTap: () =>
           Navigator.of(dialogContext).pop(ImageSource.gallery),
      ),
     ],
    ),
    actions: [
     TextButton(
      onPressed: () => Navigator.of(dialogContext).pop(null),
      child: const Text('Annuler'),
     ),
    ],
   );
  },
 );

 if (source == null || !mounted) return;

 try {
  final XFile? pickedFile = await picker.pickImage(
   source: source,
   maxWidth: 1024,
   maxHeight: 1024,
   imageQuality: 80,
  );

  if (pickedFile != null && mounted) {
   setState(() {
    _photoLocalPath = pickedFile.path;
   });
   ScaffoldMessenger.of(context).showSnackBar(
    const SnackBar(
     content: Text('□ Photo ajoutée avec succès'),
     backgroundColor: AppTheme.successColor,
     duration: Duration(seconds: 2),
    ),
   );
  }
 } catch (e) {
  if (mounted) {
   ScaffoldMessenger.of(context).showSnackBar(
    SnackBar(
     content: Text('□ Erreur: $e'),
     backgroundColor: AppTheme.errorColor,
    ),
   );
  }
 }
}

void _removePhoto() {
 showDialog(
  context: context,
```

```
      builder: (context) => AlertDialog(
        title: const Text('Supprimer la photo ?'),
        content: const Text('Voulez-vous vraiment supprimer cette photo ?'),
        actions: [
          TextButton(
            onPressed: () => Navigator.pop(context),
            child: const Text('Annuler'),
          ),
          ElevatedButton(
            style:
                ElevatedButton.styleFrom(backgroundColor: AppTheme.errorColor),
            onPressed: () {
              setState(() {
                _photoLocalPath = null;
                _photoUrl = null;
              });
              Navigator.pop(context);
              ScaffoldMessenger.of(context).showSnackBar(
                const SnackBar(
                  content: Text('□□ Photo supprimée'),
                  backgroundColor: AppTheme.errorColor,
                  duration: Duration(seconds: 2),
                ),
              );
            },
            child: const Text('Supprimer'),
          ),
        ],
      ),
    );
  }

  Future<void> _selectDate(bool isPlannedDate) async {
    final DateTime? picked = await showDatePicker(
      context: context,
      initialDate: isPlannedDate
          ? (_plannedDate ?? DateTime.now())
          : (_doneDate ?? DateTime.now()),
      firstDate: DateTime(2020),
      lastDate: DateTime(2030),
    );

    if (picked != null) {
      setState(() {
        if (isPlannedDate) {
          _plannedDate = picked;
        } else {
          _doneDate = picked;
        }
      });
    }
  }

  Future<void> _saveTask() async {
    if (!_formKey.currentState!.validate()) return;

    if (_selectedImmeuble == null || _selectedImmeuble!.isEmpty) {
      ScaffoldMessenger.of(context).showSnackBar(
        const SnackBar(
          content: Text('□ Veuillez sélectionner un immeuble'),
          backgroundColor: AppTheme.errorColor,
        ),
      );
      return;
    }

    setState(() => _isSaving = true);

    try {
      final currentUser = _auth.currentUser;
      String taskId = widget.task?.id ?? const Uuid().v4();

      int? taskNumber = widget.task?.taskNumber;
      if (!_isEditing) {
        taskNumber = await _localDb.getNextTaskNumber();
      }

      // Upload de la photo si nouvelle
      String photoUrl = _photoUrl ?? '';
      if (_photoLocalPath != null &&
          _photoLocalPath!.isNotEmpty &&
          _photoLocalPath != widget.task?.photoLocalPath) {
        try {
          if (await SyncService().hasConnection()) {
            photoUrl = await _supabase.uploadPhoto(_photoLocalPath!, taskId);
          }
        } catch (e) {
          // On garde le chemin local
        }
      }

      // Si photo supprimée
      if (_photoLocalPath == null && _photoUrl == null) {
        photoUrl = '';
      }

      final task = TaskModel(
        id: taskId,
        taskNumber: taskNumber,
        createdAt: widget.task?.createdAt ?? DateTime.now(),
        immeuble: _selectedImmeuble!,
```

```
      etage: _etageController.text.trim(),
      chambre: _chambreController.text.trim(),
      description: _descriptionController.text.trim(),
      done: _done,
      doneDate: _done ? (_doneDate ?? DateTime.now()) : null,
      doneBy: _doneByController.text.trim(),
      lastModifiedBy: currentUser?.id ?? '',
      photoUrl: photoUrl,
      photoLocalPath: _photoLocalPath ?? '',
      plannedDate: _plannedDate,
      syncStatus: _isEditing ? 'pending_update' : 'pending_create',
    );

    if (_isEditing) {
      await _recordChanges(widget.task!, task, currentUser?.id ?? '',
          currentUser?.nomComplet ?? '');
      await _localDb.updateTask(task);
    } else {
      await _localDb.insertTask(task);
      await _localDb.insertHistory(TaskHistoryModel(
        taskId: taskId,
        champModifie: 'creation',
        ancienneValeur: '',
        nouvelleValeur: 'Tâche créée',
        modifiedBy: currentUser?.id ?? '',
        modifiedByName: currentUser?.nomComplet ?? '',
        syncStatus: 'pending_create',
      ));
    }

    // Notifications
    if (_done && !(widget.task?.done ?? false)) {
      final users = await _localDb.getAllUsers();
      for (var user in users) {
        if (user.isAdmin && !user.archived) {
          await NotificationService().notifyTaskDone(
            user.id,
            task.description,
            _doneByController.text.trim(),
          );
        }
      }
    }

    if (mounted) {
      ScaffoldMessenger.of(context).showSnackBar(
        SnackBar(
          content: Text(
              _isEditing ? '□ Tâche modifiée' : '□ Tâche #$taskNumber créée'),
          backgroundColor: AppTheme.successColor,
        ),
      );
      Navigator.pop(context, true);
    }
  } catch (e) {
    if (mounted) {
      ScaffoldMessenger.of(context).showSnackBar(
        SnackBar(
          content: Text('□ Erreur: $e'),
          backgroundColor: AppTheme.errorColor,
        ),
      );
    }
  } finally {
    if (mounted) setState(() => _isSaving = false);
  }
}

Future<void> _recordChanges(TaskModel oldTask, TaskModel newTask,
    String modifiedBy, String modifiedByName) async {
  final changes = <MapEntry<String, List<String>>>[];

  if (oldTask.immeuble != newTask.immeuble) {
    changes.add(MapEntry('immeuble', [oldTask.immeuble, newTask.immeuble]));
  }
  if (oldTask.etage != newTask.etage) {
    changes.add(MapEntry('etage', [oldTask.etage, newTask.etage]));
  }
  if (oldTask.chambre != newTask.chambre) {
    changes.add(MapEntry('chambre', [oldTask.chambre, newTask.chambre]));
  }
  if (oldTask.description != newTask.description) {
    changes.add(
        MapEntry('description', [oldTask.description, newTask.description]));
  }
  if (oldTask.done != newTask.done) {
    changes.add(MapEntry(
        'done', [oldTask.done.toString(), newTask.done.toString()]));
  }
  if (oldTask.doneBy != newTask.doneBy) {
    changes.add(MapEntry('done_by', [oldTask.doneBy, newTask.doneBy]));
  }
  if (oldTask.photoUrl != newTask.photoUrl) {
    changes.add(MapEntry('photo_url', [
      oldTask.photoUrl.isNotEmpty ? 'Photo existante' : '',
      newTask.photoUrl.isNotEmpty ? 'Nouvelle photo' : 'Photo supprimée'
    ]));
  }
  if (oldTask.plannedDate?.toString() != newTask.plannedDate?.toString()) {
    changes.add(MapEntry('planned_date', [
      oldTask.plannedDate?.toString() ?? '',
```

```
      newTask.plannedDate?.toString() ?? ''
    ]));
  }

  for (var change in changes) {
    await _localDb.insertHistory(TaskHistoryModel(
      taskId: oldTask.id,
      champModifie: change.key,
      ancienneValeur: change.value[0],
      nouvelleValeur: change.value[1],
      modifiedBy: modifiedBy,
      modifiedByName: modifiedByName,
      syncStatus: 'pending_create',
    ));
  }
}

bool get _hasPhoto {
  return (_photoLocalPath != null && _photoLocalPath!.isNotEmpty) ||
      (_photoUrl != null && _photoUrl!.isNotEmpty);
}

@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: Text(_isEditing ? 'Modifier la tâche' : 'Nouvelle tâche'),
    ),
    body: SingleChildScrollView(
      padding: const EdgeInsets.all(16),
      child: Form(
        key: _formKey,
        child: Column(
          crossAxisAlignment: CrossAxisAlignment.start,
          children: [
            // ============================================
            // IMMEUBLE — Liste déroulante
            // ============================================
            Row(
              crossAxisAlignment: CrossAxisAlignment.start,
              children: [
                Expanded(
                  child: DropdownButtonFormField<String>(
                    value: _selectedImmeuble,
                    decoration: const InputDecoration(
                      labelText: 'Immeuble *',
                      prefixIcon: Icon(Icons.apartment),
                    ),
                    isExpanded: true,
                    hint: const Text('Sélectionner un immeuble'),
                    items: _immeubles.map((immeuble) {
                      return DropdownMenuItem<String>(
                        value: immeuble.nom,
                        child: Text(immeuble.nom),
                      );
                    }).toList(),
                    onChanged: (value) {
                      setState(() {
                        _selectedImmeuble = value;
                      });
                    },
                    validator: (value) {
                      if (value == null || value.isEmpty) {
                        return 'Veuillez sélectionner un immeuble';
                      }
                      return null;
                    },
                  ),
                ),
                const SizedBox(width: 8),
                // Bouton ajouter un immeuble
                Padding(
                  padding: const EdgeInsets.only(top: 8),
                  child: IconButton(
                    onPressed: _showAddImmeubleDialog,
                    icon: const Icon(Icons.add_circle,
                        color: AppTheme.secondaryColor, size: 32),
                    tooltip: 'Ajouter un immeuble',
                  ),
                ),
              ],
            ),
            const SizedBox(height: 16),

            // Étage et chambre
            Row(
              children: [
                Expanded(
                  child: AppTextField(
                    controller: _etageController,
                    labelText: 'Étage',
                    prefixIcon: const Icon(Icons.layers),
                  ),
                ),
                const SizedBox(width: 12),
                Expanded(
                  child: AppTextField(
                    controller: _chambreController,
                    labelText: 'Chambre',
                    prefixIcon: const Icon(Icons.door_front_door),
                  ),
```

```
    ),
  ],
),
const SizedBox(height: 16),

// Description
AppTextField(
  controller: _descriptionController,
  maxLines: 4,
  labelText: 'Description de la tâche *',
  prefixIcon: const Icon(Icons.description),
  validator: (value) {
    if (value == null || value.trim().isEmpty) {
      return 'Veuillez entrer une description';
    }
    return null;
  },
),
const SizedBox(height: 16),

// Date planifiée
Card(
  child: ListTile(
    leading: const Icon(Icons.calendar_today,
        color: AppTheme.warningColor),
    title: const Text('Date planifiée'),
    subtitle: Text(
      _plannedDate != null
          ? DateFormat('dd/MM/yyyy').format(_plannedDate!)
          : 'Non définie',
    ),
    trailing: Row(
      mainAxisSize: MainAxisSize.min,
      children: [
        IconButton(
          icon: const Icon(Icons.edit_calendar),
          onPressed: () => _selectDate(true),
        ),
        if (_plannedDate != null)
          IconButton(
            icon: const Icon(Icons.clear,
                color: AppTheme.errorColor),
            onPressed: () {
              setState(() => _plannedDate = null);
            },
          ),
      ],
    ),
  ),
),
const SizedBox(height: 16),

// Statut fait/pas fait
Card(
  child: SwitchListTile(
    secondary: Icon(
      _done ? Icons.check_circle : Icons.pending,
      color:
          _done ? AppTheme.successColor : AppTheme.warningColor,
      size: 32,
    ),
    title: Text(
      _done ? 'Tâche terminée' : 'Tâche en cours',
      style: const TextStyle(fontWeight: FontWeight.w600),
    ),
    value: _done,
    onChanged: (value) {
      setState(() {
        _done = value;
        if (value) {
          _doneDate = DateTime.now();
          _doneByController.text =
              _auth.currentUser?.nomComplet ?? '';
        } else {
          _doneDate = null;
        }
      });
    },
  ),
),

// Si fait : date, exécutant et photo
if (_done) ...[
  const SizedBox(height: 12),
  Card(
    child: ListTile(
      leading: const Icon(Icons.event_available,
          color: AppTheme.successColor),
      title: const Text('Date d\'exécution'),
      subtitle: Text(
        _doneDate != null
            ? DateFormat('dd/MM/yyyy').format(_doneDate!)
            : 'Aujourd\'hui',
      ),
      trailing: IconButton(
        icon: const Icon(Icons.edit_calendar),
        onPressed: () => _selectDate(false),
      ),
    ),
  ),
  const SizedBox(height: 12),
```

```
AppTextField(
  controller: _doneByController,
  labelText: 'Exécutant',
  prefixIcon: const Icon(Icons.person),
),
const SizedBox(height: 12),

// ============================================
// PHOTO avec bouton supprimer
// ============================================
Card(
  child: Column(
    children: [
      ListTile(
        leading: const Icon(Icons.camera_alt,
            color: AppTheme.primaryColor),
        title: const Text('Photo du travail'),
        subtitle: Text(
            _hasPhoto ? 'Photo ajoutée' : 'Optionnel'),
        trailing: ElevatedButton.icon(
          onPressed: _pickImage,
          icon: Icon(
            _hasPhoto ? Icons.change_circle : Icons.add_a_photo,
            size: 18,
          ),
          label:
              Text(_hasPhoto ? 'Changer' : 'Ajouter'),
        ),
      ),

      // Affichage de la photo locale
      if (_photoLocalPath != null &&
          _photoLocalPath!.isNotEmpty) ...[
        Padding(
          padding: const EdgeInsets.fromLTRB(12, 0, 12, 12),
          child: Stack(
            children: [
              ClipRRect(
                borderRadius: BorderRadius.circular(8),
                child: Image.file(
                  File(_photoLocalPath!),
                  height: 200,
                  width: double.infinity,
                  fit: BoxFit.cover,
                ),
              ),
              // Bouton supprimer en bas à droite
              Positioned(
                bottom: 8,
                right: 8,
                child: GestureDetector(
                  onTap: _removePhoto,
                  child: Container(
                    padding: const EdgeInsets.all(6),
                    decoration: BoxDecoration(
                      color: AppTheme.errorColor,
                      shape: BoxShape.circle,
                      boxShadow: [
                        BoxShadow(
                          color: Colors.black
                              .withValues(alpha: 77),
                          blurRadius: 4,
                          offset: const Offset(0, 2),
                        ),
                      ],
                    ),
                    child: const Icon(
                      Icons.close,
                      color: Colors.white,
                      size: 20,
                    ),
                  ),
                ),
              ),
            ],
          ),
        ),
      ]
      // Affichage de la photo depuis URL
      else if (_photoUrl != null &&
          _photoUrl!.isNotEmpty) ...[
        Padding(
          padding: const EdgeInsets.fromLTRB(12, 0, 12, 12),
          child: Stack(
            children: [
              ClipRRect(
                borderRadius: BorderRadius.circular(8),
                child: Image.network(
                  _photoUrl!,
                  height: 200,
                  width: double.infinity,
                  fit: BoxFit.cover,
                  errorBuilder: (_, __, ___) => const SizedBox(
                    height: 200,
                    child: Center(
                      child: Icon(Icons.broken_image,
                          size: 60,
                          color: AppTheme.textSecondary),
                    ),
                  ),
                ),
```

```
                        ),
                        // Bouton supprimer en bas à droite
                        Positioned(
                          bottom: 8,
                          right: 8,
                          child: GestureDetector(
                            onTap: _removePhoto,
                            child: Container(
                              padding: const EdgeInsets.all(6),
                              decoration: BoxDecoration(
                                color: AppTheme.errorColor,
                                shape: BoxShape.circle,
                                boxShadow: [
                                  BoxShadow(
                                    color: Colors.black
                                        .withValues(alpha: 77),
                                    blurRadius: 4,
                                    offset: const Offset(0, 2),
                                  ),
                                ],
                              ),
                              child: const Icon(
                                Icons.close,
                                color: Colors.white,
                                size: 20,
                              ),
                            ),
                          ),
                        ),
                      ],
                    ),
                  ),
                ),
              ],
            ],
          ),
        ),
      ],

      const SizedBox(height: 32),

      // Bouton sauvegarder
      SizedBox(
        width: double.infinity,
        height: 50,
        child: ElevatedButton.icon(
          onPressed: _isSaving ? null : _saveTask,
          icon: _isSaving
              ? const SizedBox(
                  width: 20,
                  height: 20,
                  child: CircularProgressIndicator(
                    color: Colors.white,
                    strokeWidth: 2,
                  ),
                )
              : const Icon(Icons.save),
          label: Text(_isSaving
              ? 'Enregistrement...'
              : _isEditing
                  ? 'Modifier la tâche'
                  : 'Créer la tâche'),
        ),
      ),
      const SizedBox(height: 20),
    ],
  ),
),
),
);
}
}
```

dart file:<task_history_screen.dart>

```
// lib/screens/task_history_screen.dart
// ═══════════════════════════════════════════
// ÉCRAN HISTORIQUE DES MODIFICATIONS D'UNE TÂCHE
// ═══════════════════════════════════════════

import 'package:flutter/material.dart';
import 'package:intl/intl.dart';
import '../models/task_history_model.dart';
import '../services/local_db_service.dart';
import '../services/supabase_service.dart';
import '../services/sync_service.dart';
import '../utils/theme.dart';

class TaskHistoryScreen extends StatefulWidget {
  final String taskId;

  const TaskHistoryScreen({super.key, required this.taskId});

  @override
  State<TaskHistoryScreen> createState() => _TaskHistoryScreenState();
}

class _TaskHistoryScreenState extends State<TaskHistoryScreen> {
  List<TaskHistoryModel> _history = [];
  bool _isLoading = true;
```

```
@override
void initState() {
  super.initState();
  _loadHistory();
}

Future<void> _loadHistory() async {
  setState(() => _isLoading = true);

  List<TaskHistoryModel> history = [];

  // Charger depuis le local
  history = await LocalDbService().getHistoryForTask(widget.taskId);

  // Si connecté, aussi charger depuis le serveur
  if (await SyncService().hasConnection()) {
    try {
      final serverHistory =
          await SupabaseService().getHistoryForTask(widget.taskId);
      // Fusionner : garder les entrées serveur + locales non synchronisées
      final localPending =
          history.where((h) => h.syncStatus != 'synced').toList();
      history = [...serverHistory, ...localPending];
    } catch (e) {
      // Utiliser les données locales
    }
  }

  // Trier par date décroissante
  history.sort((a, b) => b.modifiedAt.compareTo(a.modifiedAt));

  if (mounted) {
    setState(() {
      _history = history;
      _isLoading = false;
    });
  }
}

@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: const Text('Historique des modifications'),
    ),
    body: _isLoading
        ? const Center(child: CircularProgressIndicator())
        : _history.isEmpty
            ? Center(
                child: Column(
                  mainAxisAlignment: MainAxisAlignment.center,
                  children: [
                    Icon(Icons.history,
                        size: 80,
                        color: AppTheme.textSecondary.withOpacity(0.3)),
                    const SizedBox(height: 16),
                    const Text(
                      'Aucune modification enregistrée',
                      style: TextStyle(
                        fontSize: 18,
                        color: AppTheme.textSecondary,
                      ),
                    ),
                  ],
                ),
              )
            : RefreshIndicator(
                onRefresh: _loadHistory,
                child: ListView.builder(
                  padding: const EdgeInsets.all(12),
                  itemCount: _history.length,
                  itemBuilder: (context, index) {
                    final entry = _history[index];
                    return _buildHistoryTile(entry);
                  },
                ),
              ),
  );
}

Widget _buildHistoryTile(TaskHistoryModel entry) {
  return Card(
    margin: const EdgeInsets.only(bottom: 8),
    child: Padding(
      padding: const EdgeInsets.all(14),
      child: Column(
        crossAxisAlignment: CrossAxisAlignment.start,
        children: [
          // Date et auteur
          Row(
            mainAxisAlignment: MainAxisAlignment.spaceBetween,
            children: [
              Text(
                DateFormat('dd/MM/yyyy à HH:mm').format(entry.modifiedAt),
                style: const TextStyle(
                  fontSize: 13,
                  fontWeight: FontWeight.w600,
                  color: AppTheme.primaryColor,
                ),
              ),
              if (entry.syncStatus != 'synced')
```

```
                const Icon(Icons.cloud_off,
                    size: 16, color: AppTheme.warningColor),
              ],
            ),
            const SizedBox(height: 4),
            Text(
              'Par : ${entry.modifiedByName.isNotEmpty ? entry.modifiedByName : "Inconnu"}',
              style: const TextStyle(
                fontSize: 12,
                color: AppTheme.textSecondary,
              ),
            ),
            const Divider(height: 16),

            // Champ modifié
            Row(
              children: [
                Container(
                  padding:
                      const EdgeInsets.symmetric(horizontal: 8, vertical: 4),
                  decoration: BoxDecoration(
                    color: AppTheme.primaryColor.withOpacity(0.1),
                    borderRadius: BorderRadius.circular(6),
                  ),
                  child: Text(
                    entry.champLabel,
                    style: const TextStyle(
                      fontSize: 13,
                      fontWeight: FontWeight.w600,
                      color: AppTheme.primaryColor,
                    ),
                  ),
                ),
              ],
            ),
            const SizedBox(height: 8),

            // Ancienne valeur
            if (entry.ancienneValeur.isNotEmpty) ...[
              Row(
                crossAxisAlignment: CrossAxisAlignment.start,
                children: [
                  const Icon(Icons.remove_circle_outline,
                      size: 16, color: AppTheme.errorColor),
                  const SizedBox(width: 6),
                  Expanded(
                    child: Text(
                      entry.ancienneValeur,
                      style: TextStyle(
                        fontSize: 14,
                        color: AppTheme.errorColor.withOpacity(0.8),
                        decoration: TextDecoration.lineThrough,
                      ),
                    ),
                  ),
                ],
              ),
              const SizedBox(height: 4),
            ],

            // Nouvelle valeur
            Row(
              crossAxisAlignment: CrossAxisAlignment.start,
              children: [
                const Icon(Icons.add_circle_outline,
                    size: 16, color: AppTheme.successColor),
                const SizedBox(width: 6),
                Expanded(
                  child: Text(
                    entry.nouvelleValeur,
                    style: const TextStyle(
                      fontSize: 14,
                      color: AppTheme.successColor,
                      fontWeight: FontWeight.w500,
                    ),
                  ),
                ),
              ],
            ),
          ],
        ),
      ),
    );
  }
}
```

dart file:<task_list_screen.dart>

```
// lib/screens/task_list_screen.dart
// ============================================
// ÉCRAN LISTE DES TÂCHES AVEC FILTRES
// ============================================
import 'package:flutter/material.dart';
import '../models/task_model.dart';
import '../models/immeuble_model.dart';
import '../services/local_db_service.dart';
import '../services/supabase_service.dart';
import '../services/sync_service.dart';
import '../services/auth_service.dart';
import '../utils/theme.dart';
```

```dart
import '../widgets/app_drawer.dart';
import '../widgets/task_card.dart';
import 'task_form_screen.dart';
import 'task_detail_screen.dart';

class TaskListScreen extends StatefulWidget {
  const TaskListScreen({super.key});

  @override
  State<TaskListScreen> createState() => _TaskListScreenState();
}

class _TaskListScreenState extends State<TaskListScreen> {
  final LocalDbService _localDb = LocalDbService();
  final AuthService _auth = AuthService();

  List<TaskModel> _allTasks = [];
  List<TaskModel> _filteredTasks = [];
  bool _isLoading = true;

  // Liste des immeubles
  List<ImmeubleModel> _immeubles = [];
  String? _selectedImmeuble;

  // Filtres
  String _statusFilter = 'en_cours';

  @override
  void initState() {
    super.initState();
    _loadImmeubles();
    _loadTasks();
  }

  Future<void> _loadImmeubles() async {
    final immeubles = await _localDb.getActiveImmeubles();
    if (mounted) {
      setState(() {
        _immeubles = immeubles;
      });
    }
  }

  Future<void> _loadTasks() async {
    setState(() => _isLoading = true);
    final tasks = await _localDb.getActiveTasks();
    if (mounted) {
      setState(() {
        _allTasks = tasks;
        _isLoading = false;
      });
      _applyFilters();
    }
  }

  void _applyFilters() {
    setState(() {
      _filteredTasks = _allTasks.where((task) {
        bool statusMatch = true;
        if (_statusFilter == 'en_cours') {
          statusMatch = !task.done;
        } else if (_statusFilter == 'terminee') {
          statusMatch = task.done;
        }

        bool immeubleMatch = true;
        if (_selectedImmeuble != null &&
            _selectedImmeuble!.isNotEmpty) {
          immeubleMatch = task.immeuble == _selectedImmeuble;
        }

        return statusMatch && immeubleMatch;
      }).toList();
    });
  }

  void _showDeleteConfirmation(TaskModel task) {
    showDialog(
      context: context,
      builder: (context) => AlertDialog(
        title: const Text('Supprimer la tâche ?'),
        content: Text(
            'Voulez-vous vraiment supprimer la tâche ${task.displayNumber} ?\n\n"${task.description}"'),
        actions: [
          TextButton(
            onPressed: () => Navigator.pop(context),
            child: const Text('Annuler'),
          ),
          ElevatedButton(
            style: ElevatedButton.styleFrom(
                backgroundColor: AppTheme.errorColor),
            onPressed: () async {
              Navigator.pop(context);
              await _deleteTask(task);
            },
            child: const Text('Supprimer'),
          ),
        ],
      ),
    );
  }
```

```
Future<void> _deleteTask(TaskModel task) async {
  final updatedTask = task.copyWith(
    deleted: true,
    syncStatus: 'pending_update',
    lastModifiedBy: _auth.currentUser?.id ?? '',
  );
  await _localDb.updateTask(updatedTask);
  await _loadTasks();
  if (mounted) {
    ScaffoldMessenger.of(context).showSnackBar(
      const SnackBar(
        content: Text('□□ Tâche supprimée'),
        backgroundColor: AppTheme.errorColor,
      ),
    );
  }
}

void _showArchiveConfirmation(TaskModel task) {
  showDialog(
    context: context,
    builder: (context) => AlertDialog(
      title: const Text('Archiver la tâche ?'),
      content: Text(
          'Voulez-vous archiver la tâche ${task.displayNumber} ?\n\n"${task.description}"'),
      actions: [
        TextButton(
          onPressed: () => Navigator.pop(context),
          child: const Text('Annuler'),
        ),
        ElevatedButton(
          onPressed: () async {
            Navigator.pop(context);
            await _archiveTask(task);
          },
          child: const Text('Archiver'),
        ),
      ],
    ),
  );
}

Future<void> _archiveTask(TaskModel task) async {
  final updatedTask = task.copyWith(
    archived: true,
    syncStatus: 'pending_update',
    lastModifiedBy: _auth.currentUser?.id ?? '',
  );

  await _localDb.updateTask(updatedTask);

  if (await SyncService().hasConnection()) {
    try {
      await SupabaseService()
          .upsertTask(updatedTask.copyWith(syncStatus: 'synced'));
      await _localDb.deleteTask(task.id);
    } catch (e) {
      // En cas d'erreur, elle reste en local avec pending_update
    }
  }

  await _loadTasks();

  if (mounted) {
    ScaffoldMessenger.of(context).showSnackBar(
      const SnackBar(
        content: Text('□ Tâche archivée'),
        backgroundColor: AppTheme.archiveColor,
      ),
    );
  }
}

// ============================================
// WIDGET CHIP DE FILTRE STATUT
// ============================================
Widget _buildStatusChip({
  required String label,
  required String value,
  required IconData icon,
  required Color color,
}) {
  final bool isSelected = _statusFilter == value;

  return FilterChip(
    label: Row(
      mainAxisSize: MainAxisSize.min,
      children: [
        Icon(icon,
            size: 16,
            color: isSelected ? Colors.white : color),
        const SizedBox(width: 4),
        Text(
          label,
          style: TextStyle(
            fontSize: 12,
            fontWeight: FontWeight.w600,
            color: isSelected ? Colors.white : color,
          ),
        ),
```

```
      ],
    ),
    selected: isSelected,
    onSelected: (_) {
      setState(() {
        _statusFilter = value;
      });
      _applyFilters();
    },
    backgroundColor: color.withValues(alpha: 25),
    selectedColor: color,
    showCheckmark: false,
    padding: const EdgeInsets.symmetric(
        horizontal: 4, vertical: 0),
    materialTapTargetSize: MaterialTapTargetSize.shrinkWrap,
    visualDensity: VisualDensity.compact,
  );
}

// ============================================
// MENU ACTIONS SUR UNE TÂCHE (APPUI LONG)
// ============================================
void _showTaskActions(TaskModel task, bool isAdmin) {
  showModalBottomSheet(
    context: context,
    shape: const RoundedRectangleBorder(
      borderRadius:
          BorderRadius.vertical(top: Radius.circular(16)),
    ),
    builder: (context) => Padding(
      padding: const EdgeInsets.symmetric(vertical: 16),
      child: Column(
        mainAxisSize: MainAxisSize.min,
        children: [
          Container(
            width: 40,
            height: 4,
            decoration: BoxDecoration(
              color: Colors.grey[300],
              borderRadius: BorderRadius.circular(2),
            ),
          ),
          const SizedBox(height: 16),
          Text(
            'Tâche ${task.displayNumber}',
            style: const TextStyle(
                fontSize: 18, fontWeight: FontWeight.bold),
          ),
          const SizedBox(height: 8),
          ListTile(
            leading: const Icon(Icons.edit,
                color: AppTheme.primaryColor),
            title: const Text('Modifier'),
            onTap: () {
              Navigator.pop(context);
              Navigator.push(
                context,
                MaterialPageRoute(
                  builder: (_) =>
                      TaskFormScreen(task: task),
                ),
              ).then((_) => _loadTasks());
            },
          ),
          ListTile(
            leading: const Icon(Icons.history,
                color: AppTheme.warningColor),
            title: const Text('Voir l\'historique'),
            onTap: () {
              Navigator.pop(context);
              Navigator.push(
                context,
                MaterialPageRoute(
                  builder: (_) => TaskDetailScreen(
                      task: task, showHistory: true),
                ),
              );
            },
          ),
          if (isAdmin) ...[
            ListTile(
              leading: const Icon(Icons.archive,
                  color: AppTheme.archiveColor),
              title: const Text('Archiver'),
              onTap: () {
                Navigator.pop(context);
                _showArchiveConfirmation(task);
              },
            ),
          ],
          ListTile(
            leading: const Icon(Icons.delete,
                color: AppTheme.errorColor),
            title: const Text('Supprimer'),
            onTap: () {
              Navigator.pop(context);
              _showDeleteConfirmation(task);
            },
          ),
        ],
      ),
```

```
    ),
  );
}

@override
Widget build(BuildContext context) {
  final isAdmin = _auth.isAdmin;

  return Scaffold(
    appBar: AppBar(
      title: const Text('Liste des tâches'),
    ),
    drawer: const AppDrawer(),
    floatingActionButton: FloatingActionButton(
      onPressed: () {
        Navigator.push(
          context,
          MaterialPageRoute(
              builder: (_) => const TaskFormScreen()),
        ).then((_) {
          _loadImmeubles();
          _loadTasks();
        });
      },
      child: const Icon(Icons.add),
    ),
    body: Column(
      children: [
        // ============================================
        // ZONE DE FILTRES
        // ============================================
        Container(
          padding:
              const EdgeInsets.fromLTRB(12, 12, 12, 4),
          color: AppTheme.backgroundColor,
          child: Column(
            crossAxisAlignment: CrossAxisAlignment.start,
            children: [
              // Filtre par statut
              Row(
                children: [
                  const Icon(Icons.filter_list,
                      size: 20,
                      color: AppTheme.textSecondary),
                  const SizedBox(width: 8),
                  Expanded(
                    child: SingleChildScrollView(
                      scrollDirection: Axis.horizontal,
                      child: Row(
                        children: [
                          _buildStatusChip(
                            label: 'Actives',
                            value: 'en_cours',
                            icon: Icons.pending,
                            color: AppTheme.warningColor,
                          ),
                          const SizedBox(width: 6),
                          _buildStatusChip(
                            label: 'Terminées',
                            value: 'terminee',
                            icon: Icons.check_circle,
                            color: AppTheme.successColor,
                          ),
                          const SizedBox(width: 6),
                          _buildStatusChip(
                            label: 'Toutes',
                            value: 'toutes',
                            icon: Icons.list,
                            color: AppTheme.primaryColor,
                          ),
                        ],
                      ),
                    ),
                  ),
                ],
              ),
              const SizedBox(height: 8),

              // ============================================
              // FILTRE PAR IMMEUBLE — Liste déroulante
              // ============================================
              SizedBox(
                height: 48,
                child: DropdownButtonFormField<String>(
                  value: _selectedImmeuble,
                  decoration: InputDecoration(
                    hintText: 'Tous les immeubles',
                    hintStyle:
                        const TextStyle(fontSize: 13),
                    prefixIcon: const Icon(
                        Icons.apartment,
                        size: 20),
                    suffixIcon: _selectedImmeuble != null
                        ? IconButton(
                            icon: const Icon(Icons.clear,
                                size: 18),
                            padding: EdgeInsets.zero,
                            onPressed: () {
                              setState(() =>
                                  _selectedImmeuble =
                                      null);
```

```
                    _applyFilters();
                  },
                )
              : null,
          contentPadding:
              const EdgeInsets.symmetric(
                horizontal: 12, vertical: 0),
          border: OutlineInputBorder(
            borderRadius:
                BorderRadius.circular(8),
          ),
        ),
        isExpanded: true,
        isDense: true,
        items: [
          const DropdownMenuItem<String>(
            value: null,
            child: Text('Tous les immeubles',
                style: TextStyle(fontSize: 14)),
          ),
          ..._immeubles.map((immeuble) {
            return DropdownMenuItem<String>(
              value: immeuble.nom,
              child: Text(immeuble.nom,
                  style: const TextStyle(
                      fontSize: 14)),
            );
          }),
        ],
        onChanged: (value) {
          setState(() {
            _selectedImmeuble = value;
          });
          _applyFilters();
        },
      ),
    ),
  ],
 ),
),

// Compteur de résultats
Padding(
  padding: const EdgeInsets.symmetric(
      horizontal: 16, vertical: 6),
  child: Row(
    mainAxisAlignment:
        MainAxisAlignment.spaceBetween,
    children: [
      Text(
        '${_filteredTasks.length} tâche(s)',
        style: const TextStyle(
          color: AppTheme.textSecondary,
          fontWeight: FontWeight.w600,
          fontSize: 13,
        ),
      ),
      IconButton(
        icon:
            const Icon(Icons.refresh, size: 20),
        padding: EdgeInsets.zero,
        constraints: const BoxConstraints(),
        onPressed: () {
          _loadImmeubles();
          _loadTasks();
        },
      ),
    ],
  ),
),
const Divider(height: 1),

// ==========================================
// LISTE DES TÂCHES
// ==========================================
Expanded(
  child: _isLoading
      ? const Center(
          child: CircularProgressIndicator())
      : _filteredTasks.isEmpty
          ? Center(
              child: Column(
                mainAxisAlignment:
                    MainAxisAlignment.center,
                children: [
                  Icon(
                    _statusFilter == 'en_cours'
                        ? Icons
                            .check_circle_outline
                        : _statusFilter ==
                                'terminee'
                            ? Icons
                                .pending_outlined
                            : Icons.inbox,
                    size: 80,
                    color: AppTheme.textSecondary
                        .withValues(alpha: 77),
                  ),
                  const SizedBox(height: 16),
                  Text(
                    _statusFilter == 'en_cours'
```

```
                            ? 'Aucune tâche en cours'
                            : _statusFilter ==
                                    'terminee'
                                ? 'Aucune tâche terminée'
                                : 'Aucune tâche',
                        style: const TextStyle(
                          fontSize: 18,
                          color:
                              AppTheme.textSecondary,
                        ),
                      ),
                      if (_selectedImmeuble !=
                          null) ...[
                        const SizedBox(height: 8),
                        Text(
                          'pour "$_selectedImmeuble"',
                          style: const TextStyle(
                            fontSize: 14,
                            color: AppTheme
                                .textSecondary,
                          ),
                        ),
                      ],
                    ],
                  ),
                )
              : RefreshIndicator(
                  onRefresh: _loadTasks,
                  child: ListView.builder(
                    padding: const EdgeInsets.only(
                        bottom: 80),
                    itemCount:
                        _filteredTasks.length,
                    itemBuilder: (context, index) {
                      final task =
                          _filteredTasks[index];
                      return TaskCard(
                        task: task,
                        onTap: () {
                          Navigator.push(
                            context,
                            MaterialPageRoute(
                              builder: (_) =>
                                  TaskDetailScreen(
                                      task: task),
                            ),
                          ).then((_) =>
                              _loadTasks());
                        },
                        onLongPress: () {
                          _showTaskActions(
                              task, isAdmin);
                        },
                      );
                    },
                  ),
                ),
        ),
      ],
    ),
  );
 }
}
```

dart file:<user_form_screen.dart>

```
// lib/screens/user_form_screen.dart
// ============================================
// ÉCRAN FORMULAIRE CRÉATION/MODIFICATION UTILISATEUR
// ============================================

import 'package:flutter/material.dart';
import 'package:uuid/uuid.dart';
import '../models/user_model.dart';
import '../services/local_db_service.dart';
import '../services/supabase_service.dart';
import '../services/auth_service.dart';
import '../services/sync_service.dart';
import '../utils/constants.dart';
import '../utils/theme.dart';
import '../widgets/app_text_field.dart';

class UserFormScreen extends StatefulWidget {
  final UserModel? user;

  const UserFormScreen({super.key, this.user});

  @override
  State<UserFormScreen> createState() => _UserFormScreenState();
}

class _UserFormScreenState extends State<UserFormScreen> {
  final _formKey = GlobalKey<FormState>();
  final LocalDbService _localDb = LocalDbService();
  final SupabaseService _supabase = SupabaseService();
  final AuthService _auth = AuthService();

  late TextEditingController _identifiantController;
  late TextEditingController _passwordController;
  late TextEditingController _nomController;
```

```
late TextEditingController _prenomController;
late TextEditingController _telephoneController;
late TextEditingController _emailController;
String _selectedRole = AppConstants.roleExecutant;
bool _isSaving = false;
bool _obscurePassword = true;

bool get _isEditing => widget.user != null;

@override
void initState() {
  super.initState();
  _identifiantController =
      TextEditingController(text: widget.user?.identifiant ?? '');
  _passwordController = TextEditingController();
  _nomController = TextEditingController(text: widget.user?.nom ?? '');
  _prenomController = TextEditingController(text: widget.user?.prenom ?? '');
  _telephoneController =
      TextEditingController(text: widget.user?.telephone ?? '');
  _emailController = TextEditingController(text: widget.user?.email ?? '');
  _selectedRole = widget.user?.role ?? AppConstants.roleExecutant;
}

@override
void dispose() {
  _identifiantController.dispose();
  _passwordController.dispose();
  _nomController.dispose();
  _prenomController.dispose();
  _telephoneController.dispose();
  _emailController.dispose();
  super.dispose();
}

Future<void> _saveUser() async {
  if (!_formKey.currentState!.validate()) return;

  setState(() => _isSaving = true);

  try {
    String id = widget.user?.id ?? const Uuid().v4();
    String passwordHash;

    if (_isEditing) {
      if (_passwordController.text.isEmpty) {
        passwordHash = widget.user!.motDePasseHash;
      } else {
        passwordHash = _auth.hashPassword(_passwordController.text);
      }
    } else {
      passwordHash = _auth.hashPassword(_passwordController.text);
    }

    final user = UserModel(
      id: id,
      identifiant: _identifiantController.text.trim(),
      motDePasseHash: passwordHash,
      nom: _nomController.text.trim(),
      prenom: _prenomController.text.trim(),
      telephone: _telephoneController.text.trim(),
      email: _emailController.text.trim(),
      role: _selectedRole,
      archived: widget.user?.archived ?? false,
      createdAt: widget.user?.createdAt ?? DateTime.now(),
      updatedAt: DateTime.now(),
    );

    if (_isEditing) {
      await _localDb.updateUser(user);
    } else {
      await _localDb.insertUser(user);
    }

    if (await SyncService().hasConnection()) {
      try {
        await _supabase.upsertUser(user);
      } catch (e) {
        // Sera synchronisé plus tard
      }
    }

    if (mounted) {
      ScaffoldMessenger.of(context).showSnackBar(
        SnackBar(
          content: Text(_isEditing
              ? '□ Utilisateur modifié'
              : '□ Utilisateur créé'),
          backgroundColor: AppTheme.successColor,
        ),
      );
      Navigator.pop(context, true);
    }
  } catch (e) {
    if (mounted) {
      ScaffoldMessenger.of(context).showSnackBar(
        SnackBar(
          content: Text('□ Erreur: $e'),
          backgroundColor: AppTheme.errorColor,
        ),
      );
    }
```

```
  } finally {
    if (mounted) setState(() => _isSaving = false);
  }
}

@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: Text(
          _isEditing ? 'Modifier l\'utilisateur' : 'Nouvel utilisateur'),
    ),
    body: SingleChildScrollView(
      padding: const EdgeInsets.all(16),
      child: Form(
        key: _formKey,
        child: Column(
          crossAxisAlignment: CrossAxisAlignment.start,
          children: [
            // Identifiant
            AppTextField(
              controller: _identifiantController,
              labelText: 'Identifiant *',
              prefixIcon: const Icon(Icons.person),
              helperText: 'Sera utilisé pour la connexion',
              enabled: !_isEditing,
              validator: (value) {
                if (value == null || value.trim().isEmpty) {
                  return 'Veuillez entrer un identifiant';
                }
                if (value.trim().length < 3) {
                  return 'Minimum 3 caractères';
                }
                return null;
              },
            ),
            const SizedBox(height: 16),

            // Mot de passe
            AppTextField(
              controller: _passwordController,
              obscureText: _obscurePassword,
              labelText: _isEditing
                  ? 'Nouveau mot de passe (laisser vide pour ne pas changer)'
                  : 'Mot de passe *',
              prefixIcon: const Icon(Icons.lock),
              suffixIcon: IconButton(
                icon: Icon(_obscurePassword
                    ? Icons.visibility_off
                    : Icons.visibility),
                onPressed: () {
                  setState(() => _obscurePassword = !_obscurePassword);
                },
              ),
              validator: (value) {
                if (!_isEditing && (value == null || value.isEmpty)) {
                  return 'Veuillez entrer un mot de passe';
                }
                if (value != null && value.isNotEmpty && value.length < 4) {
                  return 'Minimum 4 caractères';
                }
                return null;
              },
            ),
            const SizedBox(height: 16),

            // Nom
            AppTextField(
              controller: _nomController,
              labelText: 'Nom *',
              prefixIcon: const Icon(Icons.badge),
              validator: (value) {
                if (value == null || value.trim().isEmpty) {
                  return 'Veuillez entrer le nom';
                }
                return null;
              },
            ),
            const SizedBox(height: 16),

            // Prénom
            AppTextField(
              controller: _prenomController,
              labelText: 'Prénom *',
              prefixIcon: const Icon(Icons.badge),
              validator: (value) {
                if (value == null || value.trim().isEmpty) {
                  return 'Veuillez entrer le prénom';
                }
                return null;
              },
            ),
            const SizedBox(height: 16),

            // Téléphone
            AppTextField(
              controller: _telephoneController,
              keyboardType: TextInputType.phone,
              labelText: 'Téléphone',
              prefixIcon: const Icon(Icons.phone),
            ),
```

```
            const SizedBox(height: 16),

            // Email
            AppTextField(
              controller: _emailController,
              keyboardType: TextInputType.emailAddress,
              labelText: 'Email',
              prefixIcon: const Icon(Icons.email),
            ),
            const SizedBox(height: 16),

            // Rôle
            Card(
              child: Padding(
                padding: const EdgeInsets.all(16),
                child: Column(
                  crossAxisAlignment: CrossAxisAlignment.start,
                  children: [
                    const Text(
                      'Rôle de l\'utilisateur',
                      style: TextStyle(
                        fontSize: 16,
                        fontWeight: FontWeight.w600,
                      ),
                    ),
                    const SizedBox(height: 12),
                    RadioListTile<String>(
                      title: const Row(
                        children: [
                          Icon(Icons.engineering,
                              color: AppTheme.secondaryColor),
                          SizedBox(width: 8),
                          Text('Exécutant'),
                        ],
                      ),
                      subtitle:
                          const Text('Peut voir et modifier les tâches'),
                      value: AppConstants.roleExecutant,
                      groupValue: _selectedRole,
                      onChanged: (value) {
                        setState(() => _selectedRole = value!);
                      },
                    ),
                    RadioListTile<String>(
                      title: const Row(
                        children: [
                          Icon(Icons.admin_panel_settings,
                              color: AppTheme.primaryColor),
                          SizedBox(width: 8),
                          Text('Administrateur'),
                        ],
                      ),
                      subtitle:
                          const Text('Accès complet à l\'application'),
                      value: AppConstants.roleAdmin,
                      groupValue: _selectedRole,
                      onChanged: (value) {
                        setState(() => _selectedRole = value!);
                      },
                    ),
                  ],
                ),
              ),
            ),
            const SizedBox(height: 32),

            // Bouton sauvegarder
            SizedBox(
              width: double.infinity,
              height: 50,
              child: ElevatedButton.icon(
                onPressed: _isSaving ? null : _saveUser,
                icon: _isSaving
                    ? const SizedBox(
                        width: 20,
                        height: 20,
                        child: CircularProgressIndicator(
                          color: Colors.white,
                          strokeWidth: 2,
                        ),
                      )
                    : const Icon(Icons.save),
                label: Text(_isSaving
                    ? 'Enregistrement...'
                    : _isEditing
                        ? 'Modifier l\'utilisateur'
                        : 'Créer l\'utilisateur'),
              ),
            ),
            const SizedBox(height: 20),
          ],
        ),
      ),
    ),
  );
 }
}
```

dart file:<user_management_screen.dart>

```dart
// lib/screens/user_management_screen.dart
// ============================================
// ÉCRAN GESTION DES UTILISATEURS (ADMIN UNIQUEMENT)
// ============================================
import 'package:flutter/material.dart';
import '../models/user_model.dart';
import '../services/local_db_service.dart';
import '../services/supabase_service.dart';
import '../services/sync_service.dart';
import '../utils/theme.dart';
import '../widgets/app_drawer.dart';
import 'user_form_screen.dart';

class UserManagementScreen extends StatefulWidget {
  const UserManagementScreen({super.key});

  @override
  State<UserManagementScreen> createState() =>
      _UserManagementScreenState();
}

class _UserManagementScreenState
    extends State<UserManagementScreen> {
  List<UserModel> _users = [];
  bool _isLoading = true;
  bool _showArchived = false;

  @override
  void initState() {
    super.initState();
    _loadUsers();
  }

  Future<void> _loadUsers() async {
    setState(() => _isLoading = true);

    List<UserModel> users;
    if (_showArchived) {
      users = await LocalDbService().getAllUsers();
    } else {
      users = await LocalDbService().getActiveUsers();
    }

    if (mounted) {
      setState(() {
        _users = users;
        _isLoading = false;
      });
    }
  }

  Future<void> _toggleArchive(UserModel user) async {
    final newArchived = !user.archived;
    final action = newArchived ? 'Archiver' : 'Désarchiver';

    final confirm = await showDialog<bool>(
      context: context,
      builder: (context) => AlertDialog(
        title: Text('$action l\'utilisateur ?'),
        content: Text(
            'Voulez-vous $action ${user.nomComplet} ?'),
        actions: [
          TextButton(
            onPressed: () => Navigator.pop(context, false),
            child: const Text('Annuler'),
          ),
          ElevatedButton(
            onPressed: () => Navigator.pop(context, true),
            child: Text(action),
          ),
        ],
      ),
    );

    if (confirm != true) return;

    final updatedUser = user.copyWith(
      archived: newArchived,
      updatedAt: DateTime.now(),
    );

    await LocalDbService().updateUser(updatedUser);

    // Synchroniser si possible
    if (await SyncService().hasConnection()) {
      try {
        await SupabaseService().updateUser(updatedUser);
      } catch (e) {
        // Sera synchronisé plus tard
      }
    }

    await _loadUsers();

    if (mounted) {
      ScaffoldMessenger.of(context).showSnackBar(
        SnackBar(
          content: Text(newArchived
              ? '□ Utilisateur archivé'
              : '□ Utilisateur désarchivé'),
          backgroundColor: newArchived
```

```
            ? AppTheme.archiveColor
            : AppTheme.successColor,
      ),
    );
  }
}

@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: const Text('Gestion des utilisateurs'),
      actions: [
        FilterChip(
          label: Text(
            _showArchived ? 'Tous' : 'Actifs',
            style: const TextStyle(
                color: Colors.white, fontSize: 12),
          ),
          selected: _showArchived,
          onSelected: (value) {
            setState(() => _showArchived = value);
            _loadUsers();
          },
          backgroundColor: Colors.white24,
          selectedColor: Colors.white38,
          checkmarkColor: Colors.white,
        ),
        const SizedBox(width: 8),
      ],
    ),
    drawer: const AppDrawer(),
    floatingActionButton: FloatingActionButton(
      onPressed: () {
        Navigator.push(
          context,
          MaterialPageRoute(
              builder: (_) => const UserFormScreen()),
        ).then((_) => _loadUsers());
      },
      child: const Icon(Icons.person_add),
    ),
    body: _isLoading
        ? const Center(child: CircularProgressIndicator())
        : _users.isEmpty
            ? Center(
                child: Column(
                  mainAxisAlignment: MainAxisAlignment.center,
                  children: [
                    Icon(Icons.people_outline,
                        size: 80,
                        color: AppTheme.textSecondary
                            .withOpacity(0.3)),
                    const SizedBox(height: 16),
                    const Text(
                      'Aucun utilisateur',
                      style: TextStyle(
                        fontSize: 18,
                        color: AppTheme.textSecondary,
                      ),
                    ),
                  ],
                ),
              )
            : RefreshIndicator(
                onRefresh: _loadUsers,
                child: ListView.builder(
                  padding: const EdgeInsets.all(12),
                  itemCount: _users.length,
                  itemBuilder: (context, index) {
                    final user = _users[index];
                    return _buildUserCard(user);
                  },
                ),
              ),
  );
}

Widget _buildUserCard(UserModel user) {
  return Card(
    child: ListTile(
      leading: CircleAvatar(
        backgroundColor: user.archived
            ? AppTheme.archiveColor
            : user.isAdmin
                ? AppTheme.primaryColor
                : AppTheme.secondaryColor,
        child: Text(
          user.prenom.isNotEmpty
              ? user.prenom[0].toUpperCase()
              : '?',
          style: const TextStyle(
              color: Colors.white,
              fontWeight: FontWeight.bold),
        ),
      ),
      title: Row(
        children: [
          Text(
            user.nomComplet,
            style: TextStyle(
```

```
                fontWeight: FontWeight.w600,
                color:
                    user.archived ? AppTheme.archiveColor : null,
                decoration: user.archived
                    ? TextDecoration.lineThrough
                    : null,
              ),
            ),
            const SizedBox(width: 8),
            Container(
              padding: const EdgeInsets.symmetric(
                  horizontal: 8, vertical: 2),
              decoration: BoxDecoration(
                color: user.isAdmin
                    ? AppTheme.primaryColor.withOpacity(0.1)
                    : AppTheme.secondaryColor.withOpacity(0.1),
                borderRadius: BorderRadius.circular(10),
              ),
              child: Text(
                user.isAdmin ? 'Admin' : 'Exécutant',
                style: TextStyle(
                  fontSize: 11,
                  color: user.isAdmin
                      ? AppTheme.primaryColor
                      : AppTheme.secondaryColor,
                ),
              ),
            ),
          ],
        ),
        subtitle: Text(user.identifiant),
        trailing: PopupMenuButton<String>(
          onSelected: (value) {
            if (value == 'edit') {
              Navigator.push(
                context,
                MaterialPageRoute(
                    builder: (_) =>
                        UserFormScreen(user: user)),
              ).then((_) => _loadUsers());
            } else if (value == 'archive') {
              _toggleArchive(user);
            }
          },
          itemBuilder: (context) => [
            const PopupMenuItem(
              value: 'edit',
              child: Row(
                children: [
                  Icon(Icons.edit,
                      color: AppTheme.primaryColor),
                  SizedBox(width: 8),
                  Text('Modifier'),
                ],
              ),
            ),
            PopupMenuItem(
              value: 'archive',
              child: Row(
                children: [
                  Icon(
                    user.archived
                        ? Icons.unarchive
                        : Icons.archive,
                    color: user.archived
                        ? AppTheme.successColor
                        : AppTheme.warningColor,
                  ),
                  const SizedBox(width: 8),
                  Text(user.archived
                      ? 'Désarchiver'
                      : 'Archiver'),
                ],
              ),
            ),
          ],
        ),
      ),
    );
  }
}


***dart file:<auth_service.dart>***

// lib/services/auth_service.dart
// ============================================
// SERVICE D'AUTHENTIFICATION
// ============================================

import 'dart:convert';
import 'package:crypto/crypto.dart';
import 'package:shared_preferences/shared_preferences.dart';
import '../models/user_model.dart';
import 'local_db_service.dart';
import 'supabase_service.dart';

class AuthService {
  static final AuthService _instance = AuthService._internal();
  factory AuthService() => _instance;
  AuthService._internal();
```

```
final LocalDbService _localDb = LocalDbService();
final SupabaseService _supabase = SupabaseService();

UserModel? _currentUser;
UserModel? get currentUser => _currentUser;

// Hash du mot de passe avec SHA-256
String hashPassword(String password) {
  var bytes = utf8.encode(password);
  var digest = sha256.convert(bytes);
  return digest.toString();
}

// Connexion
Future<UserModel?> login(String identifiant, String motDePasse) async {
  String hash = hashPassword(motDePasse);

  // DEBUG : Afficher le hash pour vérification
  print('=== DEBUG LOGIN ===');
  print('Identifiant: $identifiant');
  print('Hash calculé: $hash');

  UserModel? user;

  // 1. Essayer d'abord sur le serveur Supabase
  try {
    print('Tentative de connexion à Supabase...');
    user = await _supabase.getUserByIdentifiant(identifiant);
    if (user != null) {
      print('Utilisateur trouvé sur Supabase: ${user.nomComplet}');
      print('Hash en base: ${user.motDePasseHash}');
      // Sauvegarder en local pour le mode offline
      await _localDb.insertUser(user);
    } else {
      print('Utilisateur NON trouvé sur Supabase');
    }
  } catch (e) {
    print('Erreur Supabase: $e');
    // Pas de connexion, on continue avec le local
  }

  // 2. Si pas trouvé sur le serveur, chercher en local
  if (user == null) {
    print('Recherche en local...');
    user = await _localDb.getUserByIdentifiant(identifiant);
    if (user != null) {
      print('Utilisateur trouvé en local: ${user.nomComplet}');
      print('Hash en base locale: ${user.motDePasseHash}');
    } else {
      print('Utilisateur NON trouvé en local');
    }
  }

  // 3. Vérifications
  if (user == null) {
    print('ÉCHEC: Utilisateur introuvable');
    return null;
  }

  if (user.motDePasseHash != hash) {
    print('ÉCHEC: Mot de passe incorrect');
    print('Hash attendu: ${user.motDePasseHash}');
    print('Hash fourni:  $hash');
    return null;
  }

  if (user.archived) {
    print('ÉCHEC: Utilisateur archivé');
    return null;
  }

  print('SUCCÈS: Connexion réussie pour ${user.nomComplet}');
  _currentUser = user;

  // Sauvegarder la session
  final prefs = await SharedPreferences.getInstance();
  await prefs.setString('current_user_id', user.id);
  await prefs.setString('current_user_role', user.role);

  return user;
}

// Déconnexion
Future<void> logout() async {
  _currentUser = null;
  final prefs = await SharedPreferences.getInstance();
  await prefs.remove('current_user_id');
  await prefs.remove('current_user_role');
}

// Restaurer la session au lancement
Future<UserModel?> restoreSession() async {
  final prefs = await SharedPreferences.getInstance();
  String? userId = prefs.getString('current_user_id');
  if (userId == null) return null;

  _currentUser = await _localDb.getUserById(userId);
  return _currentUser;
}
```

```
  // Vérifier si administrateur
  bool get isAdmin => _currentUser?.role == 'administrateur';

  // Vérifier si connecté
  bool get isLoggedIn => _currentUser != null;
}


***dart file:<local_db_service.dart>***

// lib/services/local_db_service.dart
// ============================================
// SERVICE DE BASE DE DONNÉES LOCALE (SQLite)
// ============================================

import 'package:sqflite/sqflite.dart';
import 'package:path/path.dart';
import '../models/user_model.dart';
import '../models/task_model.dart';
import '../models/task_history_model.dart';
import '../models/immeuble_model.dart';

class LocalDbService {
  static Database? _database;
  static final LocalDbService _instance = LocalDbService._internal();

  factory LocalDbService() => _instance;
  LocalDbService._internal();

  Future<Database> get database async {
    if (_database != null) return _database!;
    _database = await _initDB();
    return _database!;
  }

  Future<Database> _initDB() async {
    String path = join(await getDatabasesPath(), 'entretien_immeuble.db');
    return await openDatabase(
      path,
      version: 1,
      onCreate: _createDB,
    );
  }

  Future<void> _createDB(Database db, int version) async {
    // Table des utilisateurs
    await db.execute('''
      CREATE TABLE profiles (
        id TEXT PRIMARY KEY,
        identifiant TEXT UNIQUE NOT NULL,
        mot_de_passe_hash TEXT NOT NULL,
        nom TEXT NOT NULL,
        prenom TEXT NOT NULL,
        telephone TEXT DEFAULT '',
        email TEXT DEFAULT '',
        role TEXT NOT NULL DEFAULT 'executant',
        archived INTEGER DEFAULT 0,
        created_at TEXT NOT NULL,
        updated_at TEXT NOT NULL
      )
    ''');

    // Table des tâches
    await db.execute('''
      CREATE TABLE tasks (
        id TEXT PRIMARY KEY,
        task_number INTEGER,
        created_at TEXT NOT NULL,
        updated_at TEXT NOT NULL,
        immeuble TEXT NOT NULL,
        etage TEXT DEFAULT '',
        chambre TEXT DEFAULT '',
        description TEXT NOT NULL,
        done INTEGER DEFAULT 0,
        done_date TEXT,
        done_by TEXT DEFAULT '',
        last_modified_by TEXT DEFAULT '',
        photo_url TEXT DEFAULT '',
        photo_local_path TEXT DEFAULT '',
        archived INTEGER DEFAULT 0,
        planned_date TEXT,
        deleted INTEGER DEFAULT 0,
        sync_status TEXT DEFAULT 'synced'
      )
    ''');

    // Table historique des modifications
    await db.execute('''
      CREATE TABLE task_history (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        server_id INTEGER,
        task_id TEXT NOT NULL,
        champ_modifie TEXT NOT NULL,
        ancienne_valeur TEXT DEFAULT '',
        nouvelle_valeur TEXT DEFAULT '',
        modified_by TEXT DEFAULT '',
        modified_by_name TEXT DEFAULT '',
        modified_at TEXT NOT NULL,
        sync_status TEXT DEFAULT 'synced'
      )
    ''');
```

```
  // Table des immeubles
  await db.execute('''
    CREATE TABLE immeubles (
      id TEXT PRIMARY KEY,
      nom TEXT UNIQUE NOT NULL,
      adresse TEXT DEFAULT '',
      archived INTEGER DEFAULT 0,
      created_at TEXT NOT NULL
    )
  ''');

  // ============================================
  // CRÉER L'ADMINISTRATEUR PAR DÉFAUT
  // Identifiant : admin
  // Mot de passe : admin123
  // ============================================
  await db.insert('profiles', {
    'id': 'admin-default-001',
    'identifiant': 'admin',
    'mot_de_passe_hash':
        '240be518fabd2724ddb6f04eeb1da5967448d7e831c08c8fa822809f74c720a9',
    'nom': 'Administrateur',
    'prenom': 'Principal',
    'telephone': '',
    'email': '',
    'role': 'administrateur',
    'archived': 0,
    'created_at': DateTime.now().toIso8601String(),
    'updated_at': DateTime.now().toIso8601String(),
  });
}

// ============================================
// OPÉRATIONS SUR LES UTILISATEURS
// ============================================

Future<void> insertUser(UserModel user) async {
  final db = await database;
  await db.insert(
    'profiles',
    user.toMapLocal(),
    conflictAlgorithm: ConflictAlgorithm.replace,
  );
}

Future<void> updateUser(UserModel user) async {
  final db = await database;
  await db.update(
    'profiles',
    user.toMapLocal(),
    where: 'id = ?',
    whereArgs: [user.id],
  );
}

Future<UserModel?> getUserByIdentifiant(String identifiant) async {
  final db = await database;
  final results = await db.query(
    'profiles',
    where: 'identifiant = ?',
    whereArgs: [identifiant],
  );
  if (results.isEmpty) return null;
  return UserModel.fromMap(results.first);
}

Future<UserModel?> getUserById(String id) async {
  final db = await database;
  final results = await db.query(
    'profiles',
    where: 'id = ?',
    whereArgs: [id],
  );
  if (results.isEmpty) return null;
  return UserModel.fromMap(results.first);
}

Future<List<UserModel>> getAllUsers() async {
  final db = await database;
  final results = await db.query('profiles', orderBy: 'nom ASC');
  return results.map((map) => UserModel.fromMap(map)).toList();
}

Future<List<UserModel>> getActiveUsers() async {
  final db = await database;
  final results = await db.query(
    'profiles',
    where: 'archived = 0',
    orderBy: 'nom ASC',
  );
  return results.map((map) => UserModel.fromMap(map)).toList();
}

// ============================================
// OPÉRATIONS SUR LES TÂCHES
// ============================================

Future<void> insertTask(TaskModel task) async {
  final db = await database;
  await db.insert(
```

```
    'tasks',
    task.toMapLocal(),
    conflictAlgorithm: ConflictAlgorithm.replace,
  );
}

Future<void> updateTask(TaskModel task) async {
  final db = await database;
  await db.update(
    'tasks',
    task.toMapLocal(),
    where: 'id = ?',
    whereArgs: [task.id],
  );
}

Future<void> deleteTask(String taskId) async {
  final db = await database;
  await db.delete(
    'tasks',
    where: 'id = ?',
    whereArgs: [taskId],
  );
}

Future<TaskModel?> getTaskById(String id) async {
  final db = await database;
  final results = await db.query(
    'tasks',
    where: 'id = ?',
    whereArgs: [id],
  );
  if (results.isEmpty) return null;
  return TaskModel.fromMap(results.first);
}

// Tâches actives (non archivées, non supprimées)
Future<List<TaskModel>> getActiveTasks() async {
  final db = await database;
  final results = await db.query(
    'tasks',
    where: 'archived = 0 AND deleted = 0',
    orderBy: 'created_at DESC',
  );
  return results.map((map) => TaskModel.fromMap(map)).toList();
}

// Tâches en cours uniquement (non faites, non archivées, non supprimées)
Future<List<TaskModel>> getPendingTasks() async {
  final db = await database;
  final results = await db.query(
    'tasks',
    where: 'archived = 0 AND deleted = 0 AND done = 0',
    orderBy: 'created_at DESC',
  );
  return results.map((map) => TaskModel.fromMap(map)).toList();
}

// Tâches terminées (faites mais non archivées, non supprimées)
Future<List<TaskModel>> getDoneTasks() async {
  final db = await database;
  final results = await db.query(
    'tasks',
    where: 'archived = 0 AND deleted = 0 AND done = 1',
    orderBy: 'done_date DESC',
  );
  return results.map((map) => TaskModel.fromMap(map)).toList();
}

// Tâches en attente de synchronisation
Future<List<TaskModel>> getPendingSyncTasks() async {
  final db = await database;
  final results = await db.query(
    'tasks',
    where: 'sync_status != ?',
    whereArgs: ['synced'],
  );
  return results.map((map) => TaskModel.fromMap(map)).toList();
}

// Supprimer les tâches archivées du stockage local
Future<void> removeArchivedTasksLocally() async {
  final db = await database;
  await db.delete(
    'tasks',
    where: 'archived = 1 AND sync_status = ?',
    whereArgs: ['synced'],
  );
}

// Tâches planifiées pour une date
Future<List<TaskModel>> getTasksForDate(DateTime date) async {
  final db = await database;
  String dateStr = date.toIso8601String().split('T')[0];
  final results = await db.query(
    'tasks',
    where: 'planned_date = ? AND deleted = 0',
    whereArgs: [dateStr],
  );
  return results.map((map) => TaskModel.fromMap(map)).toList();
}
```

```
// Toutes les tâches avec date planifiée
Future<List<TaskModel>> getAllPlannedTasks() async {
  final db = await database;
  final results = await db.query(
    'tasks',
    where:
        'planned_date IS NOT NULL AND planned_date != "" AND deleted = 0',
    orderBy: 'planned_date ASC',
  );
  return results.map((map) => TaskModel.fromMap(map)).toList();
}

// Obtenir le prochain numéro de tâche local
Future<int> getNextTaskNumber() async {
  final db = await database;
  final result = await db.rawQuery(
    'SELECT MAX(task_number) as max_num FROM tasks',
  );
  int maxNum = 0;
  if (result.isNotEmpty && result.first['max_num'] != null) {
    maxNum = result.first['max_num'] as int;
  }
  return maxNum + 1;
}

// ============================================
// OPÉRATIONS SUR L'HISTORIQUE
// ============================================

Future<void> insertHistory(TaskHistoryModel history) async {
  final db = await database;
  await db.insert('task_history', history.toMapLocal());
}

Future<List<TaskHistoryModel>> getHistoryForTask(String taskId) async {
  final db = await database;
  final results = await db.query(
    'task_history',
    where: 'task_id = ?',
    whereArgs: [taskId],
    orderBy: 'modified_at DESC',
  );
  return results.map((map) => TaskHistoryModel.fromMap(map)).toList();
}

Future<List<TaskHistoryModel>> getPendingSyncHistory() async {
  final db = await database;
  final results = await db.query(
    'task_history',
    where: 'sync_status != ?',
    whereArgs: ['synced'],
  );
  return results.map((map) => TaskHistoryModel.fromMap(map)).toList();
}

Future<void> updateHistorySyncStatus(int id, String status,
    {int? serverId}) async {
  final db = await database;
  Map<String, dynamic> values = {'sync_status': status};
  if (serverId != null) values['server_id'] = serverId;
  await db.update(
    'task_history',
    values,
    where: 'id = ?',
    whereArgs: [id],
  );
}

// ============================================
// OPÉRATIONS SUR LES IMMEUBLES
// ============================================

Future<void> insertImmeuble(ImmeubleModel immeuble) async {
  final db = await database;
  await db.insert(
    'immeubles',
    immeuble.toMapLocal(),
    conflictAlgorithm: ConflictAlgorithm.replace,
  );
}

Future<List<ImmeubleModel>> getActiveImmeubles() async {
  final db = await database;
  final results = await db.query(
    'immeubles',
    where: 'archived = 0',
    orderBy: 'nom ASC',
  );
  return results.map((map) => ImmeubleModel.fromMap(map)).toList();
}

Future<List<ImmeubleModel>> getAllImmeubles() async {
  final db = await database;
  final results = await db.query('immeubles', orderBy: 'nom ASC');
  return results.map((map) => ImmeubleModel.fromMap(map)).toList();
}

Future<void> replaceAllImmeubles(List<ImmeubleModel> immeubles) async {
  final db = await database;
  await db.delete('immeubles');
```

```
    for (var immeuble in immeubles) {
      await db.insert('immeubles', immeuble.toMapLocal(),
          conflictAlgorithm: ConflictAlgorithm.replace);
    }
  }

  // Ajouter un immeuble s'il n'existe pas encore (par nom)
  Future<void> insertImmeubleIfNotExists(String nom) async {
    final db = await database;
    final existing = await db.query(
      'immeubles',
      where: 'nom = ?',
      whereArgs: [nom],
    );
    if (existing.isEmpty) {
      await db.insert('immeubles', {
        'id': DateTime.now().millisecondsSinceEpoch.toString(),
        'nom': nom,
        'adresse': '',
        'archived': 0,
        'created_at': DateTime.now().toIso8601String(),
      });
    }
  }

  // ==========================================
  // UTILITAIRES
  // ==========================================

  Future<void> clearAllData() async {
    final db = await database;
    await db.delete('tasks');
    await db.delete('task_history');
    await db.delete('immeubles');
  }

  Future<void> replaceAllUsers(List<UserModel> users) async {
    final db = await database;
    await db.delete('profiles');
    for (var user in users) {
      await db.insert('profiles', user.toMapLocal(),
          conflictAlgorithm: ConflictAlgorithm.replace);
    }
  }
}


***dart file:<notification_service.dart>***

// lib/services/notification_service.dart
// ==========================================
// SERVICE DE NOTIFICATIONS LOCALES
// ==========================================

import 'package:flutter_local_notifications/flutter_local_notifications.dart';
import 'supabase_service.dart';

class NotificationService {
  static final NotificationService _instance = NotificationService._internal();
  factory NotificationService() => _instance;
  NotificationService._internal();

  final FlutterLocalNotificationsPlugin _notifications =
      FlutterLocalNotificationsPlugin();
  final SupabaseService _supabase = SupabaseService();

  Future<void> initialize() async {
    const androidSettings =
        AndroidInitializationSettings('@mipmap/ic_launcher');
    const initSettings = InitializationSettings(android: androidSettings);
    await _notifications.initialize(initSettings);
  }

  // Afficher une notification locale
  Future<void> showNotification({
    required int id,
    required String title,
    required String body,
  }) async {
    const androidDetails = AndroidNotificationDetails(
      'entretien_channel',
      'Entretien Immeuble',
      channelDescription: 'Notifications de l\'application d\'entretien',
      importance: Importance.high,
      priority: Priority.high,
      showWhen: true,
    );
    const details = NotificationDetails(android: androidDetails);
    await _notifications.show(id, title, body, details);
  }

  // Vérifier les notifications du serveur pour un utilisateur
  Future<void> checkServerNotifications(String userId) async {
    try {
      final notifications =
          await _supabase.getUnreadNotifications(userId);

      for (var notif in notifications) {
        await showNotification(
          id: notif['id'] ?? 0,
          title: notif['title'] ?? 'Notification',
```

```
          body: notif['body'] ?? '',
        );
        if (notif['id'] != null) {
          await _supabase.markNotificationAsRead(notif['id']);
        }
      }
    } catch (e) {
      // Silencieux si pas de connexion
    }
  }

  // Créer une notification pour un nouvel exécutant (nouvelle tâche)
  Future<void> notifyNewTask(String executantId, String taskDescription) async {
    try {
      await _supabase.createNotification(
        targetUserId: executantId,
        title: 'Nouvelle tâche disponible',
        body: taskDescription,
        type: 'new_task',
      );
    } catch (e) {
      // Silencieux
    }
  }

  // Créer une notification pour l'admin (tâche accomplie)
  Future<void> notifyTaskDone(
      String adminId, String taskDescription, String doneBy) async {
    try {
      await _supabase.createNotification(
        targetUserId: adminId,
        title: 'Tâche accomplie',
        body: '$doneBy a terminé: $taskDescription',
        type: 'task_done',
      );
    } catch (e) {
      // Silencieux
    }
  }
}


***dart file:<supabase_service.dart>***

// lib/services/supabase_service.dart
// ============================================
// SERVICE SUPABASE (SERVEUR DISTANT)
// ============================================

import '../models/immeuble_model.dart';
import 'dart:io';
import 'package:supabase_flutter/supabase_flutter.dart';
import '../models/user_model.dart';
import '../models/task_model.dart';
import '../models/task_history_model.dart';
import '../utils/constants.dart';

class SupabaseService {
  static final SupabaseService _instance = SupabaseService._internal();
  factory SupabaseService() => _instance;
  SupabaseService._internal();

  SupabaseClient get client => Supabase.instance.client;

  // ============================================
  // UTILISATEURS
  // ============================================

  Future<List<UserModel>> getAllUsers() async {
    final response = await client
        .from(AppConstants.tableProfiles)
        .select()
        .order('nom', ascending: true);
    return (response as List).map((map) => UserModel.fromMap(map)).toList();
  }

  Future<UserModel?> getUserByIdentifiant(String identifiant) async {
    final response = await client
        .from(AppConstants.tableProfiles)
        .select()
        .eq('identifiant', identifiant)
        .maybeSingle();
    if (response == null) return null;
    return UserModel.fromMap(response);
  }

  Future<void> upsertUser(UserModel user) async {
    await client
        .from(AppConstants.tableProfiles)
        .upsert(user.toMapSupabase());
  }

  Future<void> updateUser(UserModel user) async {
    await client
        .from(AppConstants.tableProfiles)
        .update(user.toMapSupabase())
        .eq('id', user.id);
  }

  // ============================================
  // TÂCHES
```

```
// ============================================

Future<List<TaskModel>> getAllActiveTasks() async {
  final response = await client
      .from(AppConstants.tableTasks)
      .select()
      .eq('archived', false)
      .eq('deleted', false)
      .order('created_at', ascending: false);
  return (response as List).map((map) => TaskModel.fromMap(map)).toList();
}

Future<TaskModel?> getTaskById(String id) async {
  final response = await client
      .from(AppConstants.tableTasks)
      .select()
      .eq('id', id)
      .maybeSingle();
  if (response == null) return null;
  return TaskModel.fromMap(response);
}

Future<void> upsertTask(TaskModel task) async {
  await client
      .from(AppConstants.tableTasks)
      .upsert(task.toMapSupabase());
}

Future<void> deleteTaskPermanently(String taskId) async {
  await client
      .from(AppConstants.tableTasks)
      .delete()
      .eq('id', taskId);
}

// Tâches archivées avec filtres
Future<List<TaskModel>> getArchivedTasks({
  String? immeuble,
  String? etage,
  String? chambre,
  String? doneBy,
  DateTime? doneDate,
  String? orderBy,
  bool ascending = true,
}) async {
  var query = client
      .from(AppConstants.tableTasks)
      .select()
      .eq('archived', true)
      .eq('deleted', false);

  if (immeuble != null && immeuble.isNotEmpty) {
    query = query.ilike('immeuble', '%$immeuble%');
  }
  if (etage != null && etage.isNotEmpty) {
    query = query.eq('etage', etage);
  }
  if (chambre != null && chambre.isNotEmpty) {
    query = query.eq('chambre', chambre);
  }
  if (doneBy != null && doneBy.isNotEmpty) {
    query = query.ilike('done_by', '%$doneBy%');
  }
  if (doneDate != null) {
    String dateStr = doneDate.toIso8601String().split('T')[0];
    query = query.gte('done_date', '${dateStr}T00:00:00')
        .lte('done_date', '${dateStr}T23:59:59');
  }

  final response = await query.order(
    orderBy ?? 'updated_at',
    ascending: ascending,
  );

  return (response as List).map((map) => TaskModel.fromMap(map)).toList();
}

// Rapport de tâches avec filtres
Future<List<TaskModel>> getTasksReport({
  String? immeuble,
  String? etage,
  String? chambre,
  String? doneBy,
  DateTime? doneDate,
  String? status, // 'archived', 'done', 'pending'
  String? orderBy,
  bool ascending = true,
}) async {
  var query = client
      .from(AppConstants.tableTasks)
      .select()
      .eq('deleted', false);

  if (status == 'archived') {
    query = query.eq('archived', true);
  } else if (status == 'done') {
    query = query.eq('done', true).eq('archived', false);
  } else if (status == 'pending') {
    query = query.eq('done', false).eq('archived', false);
  }
```

```
  if (immeuble != null && immeuble.isNotEmpty) {
    query = query.ilike('immeuble', '%$immeuble%');
  }
  if (etage != null && etage.isNotEmpty) {
    query = query.eq('etage', etage);
  }
  if (chambre != null && chambre.isNotEmpty) {
    query = query.eq('chambre', chambre);
  }
  if (doneBy != null && doneBy.isNotEmpty) {
    query = query.ilike('done_by', '%$doneBy%');
  }
  if (doneDate != null) {
    String dateStr = doneDate.toIso8601String().split('T')[0];
    query = query.gte('done_date', '${dateStr}T00:00:00')
        .lte('done_date', '${dateStr}T23:59:59');
  }

  final response = await query.order(
    orderBy ?? 'created_at',
    ascending: ascending,
  );

  return (response as List).map((map) => TaskModel.fromMap(map)).toList();
}

// ============================================
// HISTORIQUE
// ============================================

Future<void> insertHistory(TaskHistoryModel history) async {
  await client
      .from(AppConstants.tableTaskHistory)
      .insert(history.toMapSupabase());
}

Future<List<TaskHistoryModel>> getHistoryForTask(String taskId) async {
  final response = await client
      .from(AppConstants.tableTaskHistory)
      .select()
      .eq('task_id', taskId)
      .order('modified_at', ascending: false);
  return (response as List)
      .map((map) => TaskHistoryModel.fromMap(map))
      .toList();
}

// ============================================
// NOTIFICATIONS
// ============================================

Future<void> createNotification({
  required String targetUserId,
  required String title,
  required String body,
  required String type,
  String? taskId,
}) async {
  await client.from(AppConstants.tableNotifications).insert({
    'target_user_id': targetUserId,
    'title': title,
    'body': body,
    'type': type,
    'task_id': taskId,
    'is_read': false,
  });
}

Future<List<Map<String, dynamic>>> getUnreadNotifications(
    String userId) async {
  final response = await client
      .from(AppConstants.tableNotifications)
      .select()
      .eq('target_user_id', userId)
      .eq('is_read', false)
      .order('created_at', ascending: false);
  return List<Map<String, dynamic>>.from(response);
}

Future<void> markNotificationAsRead(int notificationId) async {
  await client
      .from(AppConstants.tableNotifications)
      .update({'is_read': true}).eq('id', notificationId);
}

// ============================================
// UPLOAD DE PHOTOS
// ============================================

Future<String> uploadPhoto(String filePath, String taskId) async {
  final file = File(filePath);
  final fileName =
      'task_${taskId}_${DateTime.now().millisecondsSinceEpoch}.jpg';

  await client.storage
      .from(AppConstants.storageBucket)
      .upload(fileName, file);

  final publicUrl = client.storage
      .from(AppConstants.storageBucket)
      .getPublicUrl(fileName);
```

```
    return publicUrl;
  }

  // ============================================
  // IMMEUBLES
  // ============================================

  Future<List<ImmeubleModel>> getAllImmeubles() async {
    final response = await client
        .from('immeubles')
        .select()
        .eq('archived', false)
        .order('nom', ascending: true);
    return (response as List).map((map) => ImmeubleModel.fromMap(map)).toList();
  }

  Future<void> upsertImmeuble(ImmeubleModel immeuble) async {
    await client.from('immeubles').upsert(immeuble.toMapSupabase());
  }

  Future<void> insertImmeubleIfNotExists(String nom) async {
    final existing = await client
        .from('immeubles')
        .select()
        .eq('nom', nom)
        .maybeSingle();
    if (existing == null) {
      await client.from('immeubles').insert({
        'id': DateTime.now().millisecondsSinceEpoch.toString(),
        'nom': nom,
        'adresse': '',
        'archived': false,
      });
    }
  }
}
```

dart file:<sync_service.dart>

```
// lib/services/sync_service.dart
// ============================================
// SERVICE DE SYNCHRONISATION
// LOCAL ↔ SERVEUR DISTANT
// ============================================

import 'package:connectivity_plus/connectivity_plus.dart';
import '../models/task_model.dart';
import '../models/task_history_model.dart';
import '../models/user_model.dart';
import '../models/immeuble_model.dart';
import 'local_db_service.dart';
import 'supabase_service.dart';

class SyncService {
  static final SyncService _instance = SyncService._internal();
  factory SyncService() => _instance;
  SyncService._internal();

  final LocalDbService _localDb = LocalDbService();
  final SupabaseService _supabase = SupabaseService();

  bool _isSyncing = false;

  // Vérifier la connectivité
  Future<bool> hasConnection() async {
    final connectivityResult = await Connectivity().checkConnectivity();
    return !connectivityResult.contains(ConnectivityResult.none);
  }

  // Synchronisation complète
  Future<SyncResult> syncAll() async {
    if (_isSyncing) {
      return SyncResult(
          success: false, message: 'Synchronisation déjà en cours');
    }
    if (!await hasConnection()) {
      return SyncResult(
          success: false, message: 'Pas de connexion internet');
    }

    _isSyncing = true;
    int synced = 0;

    try {
      // 1. Envoyer les modifications locales vers le serveur
      synced += await _pushLocalChanges();

      // 2. Récupérer les données du serveur
      synced += await _pullFromServer();

      // 3. Synchroniser l'historique
      synced += await _syncHistory();

      // 4. Synchroniser les utilisateurs
      await _syncUsers();

      // 5. Synchroniser les immeubles
      await _syncImmeubles();
```

```
    // 6. Supprimer les tâches archivées du stockage local
    await _localDb.removeArchivedTasksLocally();

    _isSyncing = false;
    return SyncResult(
      success: true,
      message: '$synced éléments synchronisés',
      count: synced,
    );
  } catch (e) {
    _isSyncing = false;
    return SyncResult(
      success: false,
      message: 'Erreur de synchronisation: $e',
      count: synced,
    );
  }
}

// Envoyer les changements locaux vers Supabase
Future<int> _pushLocalChanges() async {
int count = 0;
List<TaskModel> pendingTasks = await _localDb.getPendingSyncTasks();

for (var task in pendingTasks) {
  try {
    if (task.syncStatus == 'pending_delete') {
      await _supabase.upsertTask(task.copyWith(deleted: true));
      await _localDb.deleteTask(task.id);
    } else {
      await _supabase.upsertTask(task);

      // Récupérer la tâche du serveur pour obtenir le task_number attribué
      TaskModel? serverTask = await _supabase.getTaskById(task.id);
      if (serverTask != null) {
        // Si la tâche est archivée et synchronisée, la supprimer du local
        if (serverTask.archived) {
          await _localDb.deleteTask(task.id);
        } else {
          // Mettre à jour le numéro de tâche et le statut de sync
          await _localDb.updateTask(serverTask.copyWith(
            syncStatus: 'synced',
            photoLocalPath: task.photoLocalPath,
          ));
        }
      } else {
        await _localDb.updateTask(task.copyWith(syncStatus: 'synced'));
      }

      // Aussi enregistrer l'immeuble s'il n'existe pas encore
      if (task.immeuble.isNotEmpty) {
        try {
          await _supabase.insertImmeubleIfNotExists(task.immeuble);
          await _localDb.insertImmeubleIfNotExists(task.immeuble);
        } catch (e) {
          // Ignorer les erreurs d'immeuble
        }
      }
    }
    count++;
  } catch (e) {
    // On continue avec les autres tâches
    continue;
  }
}
return count;
}

// Récupérer les tâches du serveur
Future<int> _pullFromServer() async {
  int count = 0;
  try {
    List<TaskModel> serverTasks = await _supabase.getAllActiveTasks();

    for (var serverTask in serverTasks) {
      TaskModel? localTask = await _localDb.getTaskById(serverTask.id);

      if (localTask == null) {
        // Nouvelle tâche du serveur → ajouter en local
        await _localDb
            .insertTask(serverTask.copyWith(syncStatus: 'synced'));
        count++;
      } else if (localTask.syncStatus == 'synced') {
        // Tâche déjà synchronisée → mettre à jour si plus récente
        if (serverTask.updatedAt.isAfter(localTask.updatedAt)) {
          await _localDb.updateTask(serverTask.copyWith(
            syncStatus: 'synced',
            photoLocalPath: localTask.photoLocalPath,
          ));
          count++;
        }
      }
      // Si la tâche locale a des modifications en attente, on ne la remplace pas
    }
  } catch (e) {
    rethrow;
  }
  return count;
}

// Synchroniser l'historique
```

```
  Future<int> _syncHistory() async {
    int count = 0;
    List<TaskHistoryModel> pendingHistory =
        await _localDb.getPendingSyncHistory();

    for (var history in pendingHistory) {
      try {
        await _supabase.insertHistory(history);
        if (history.id != null) {
          await _localDb.updateHistorySyncStatus(history.id!, 'synced');
        }
        count++;
      } catch (e) {
        continue;
      }
    }
    return count;
  }

  // Synchroniser les utilisateurs
  Future<void> _syncUsers() async {
    try {
      List<UserModel> serverUsers = await _supabase.getAllUsers();
      await _localDb.replaceAllUsers(serverUsers);
    } catch (e) {
      // Ignorer si pas de connexion
    }
  }

  // Synchroniser les immeubles
  Future<void> _syncImmeubles() async {
    try {
      List<ImmeubleModel> serverImmeubles =
          await _supabase.getAllImmeubles();
      await _localDb.replaceAllImmeubles(serverImmeubles);
    } catch (e) {
      // Ignorer si pas de connexion
    }
  }
}

class SyncResult {
  final bool success;
  final String message;
  final int count;

  SyncResult({
    required this.success,
    required this.message,
    this.count = 0,
  });
}


***dart file:<constants.dart>***

// lib/utils/constants.dart
// ============================================
// CONSTANTES DE L'APPLICATION
// ============================================

class AppConstants {
  // ============================================
  // CONFIGURATION SUPABASE
  // Remplacez par vos propres valeurs !
  // ============================================
  static const String supabaseUrl = 'https://ekjijwzqwllngzrmtavu.supabase.co';
  static const String supabaseAnonKey =
'eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJzdXBhYmFzZSIsInJlZiI6ImVramlqd3pxd2xsbmd6cm10YXZ1Iiwicm9sZSI6ImFub24iLCJpYXQiOjE3NzEwMDAyNjc
sImV4cCI6MjA4NjU3NjI2N30.F6qa6HGfR_URy5q4l1U77dw-Jnd5wefIbApMxyxiKRY';

  // ============================================
  // NOMS DES TABLES
  // ============================================
  static const String tableProfiles = 'profiles';
  static const String tableTasks = 'tasks';
  static const String tableTaskHistory = 'task_history';
  static const String tableNotifications = 'pending_notifications';
  static const String storageBucket = 'task-photos';

  // ============================================
  // RÔLES
  // ============================================
  static const String roleAdmin = 'administrateur';
  static const String roleExecutant = 'executant';

  // ============================================
  // STATUS DE SYNCHRONISATION
  // ============================================
  static const String syncStatusSynced = 'synced';
  static const String syncStatusPendingCreate = 'pending_create';
  static const String syncStatusPendingUpdate = 'pending_update';
  static const String syncStatusPendingDelete = 'pending_delete';
}


***dart file:<theme.dart>***

// lib/utils/theme.dart
// ============================================
// THÈME VISUEL DE L'APPLICATION
```

```dart
// Couleurs douces et police lisible
// ==========================================

import 'package:flutter/material.dart';
import 'package:google_fonts/google_fonts.dart';

class AppTheme {
  // Couleurs principales (douces et non agressives)
  static const Color primaryColor = Color(0xFF5C6BC0);     // Indigo doux
  static const Color primaryLight = Color(0xFF8E99A4);     // Gris bleuté
  static const Color secondaryColor = Color(0xFF26A69A);   // Vert teal doux
  static const Color backgroundColor = Color(0xFFF5F5F5);  // Gris très clair
  static const Color surfaceColor = Colors.white;
  static const Color errorColor = Color(0xFFE57373);       // Rouge doux
  static const Color successColor = Color(0xFF81C784);     // Vert doux
  static const Color warningColor = Color(0xFFFFB74D);     // Orange doux
  static const Color textPrimary = Color(0xFF37474F);      // Gris foncé
  static const Color textSecondary = Color(0xFF78909C);    // Gris moyen
  static const Color archiveColor = Color(0xFFB0BEC5);     // Gris clair

  static ThemeData get lightTheme {
    return ThemeData(
      useMaterial3: true,
      colorScheme: ColorScheme.fromSeed(
        seedColor: primaryColor,
        brightness: Brightness.light,
        surface: surfaceColor,
        error: errorColor,
      ),
      scaffoldBackgroundColor: backgroundColor,
      textTheme: GoogleFonts.nunitoTextTheme().apply(
        bodyColor: textPrimary,
        displayColor: textPrimary,
      ),
      appBarTheme: AppBarTheme(
        backgroundColor: primaryColor,
        foregroundColor: Colors.white,
        elevation: 2,
        centerTitle: true,
        titleTextStyle: GoogleFonts.nunito(
          fontSize: 20,
          fontWeight: FontWeight.w700,
          color: Colors.white,
        ),
      ),
      cardTheme: CardThemeData(
        elevation: 2,
        shape: RoundedRectangleBorder(borderRadius: BorderRadius.circular(12)),
        margin: const EdgeInsets.symmetric(horizontal: 12, vertical: 6),
      ),
      elevatedButtonTheme: ElevatedButtonThemeData(
        style: ElevatedButton.styleFrom(
          backgroundColor: primaryColor,
          foregroundColor: Colors.white,
          padding: const EdgeInsets.symmetric(horizontal: 24, vertical: 14),
          shape: RoundedRectangleBorder(borderRadius: BorderRadius.circular(10)),
          textStyle: GoogleFonts.nunito(
            fontSize: 16,
            fontWeight: FontWeight.w600,
          ),
        ),
      ),
      inputDecorationTheme: InputDecorationTheme(
        filled: true,
        fillColor: Colors.white,
        border: OutlineInputBorder(
          borderRadius: BorderRadius.circular(10),
          borderSide: const BorderSide(color: primaryLight),
        ),
        enabledBorder: OutlineInputBorder(
          borderRadius: BorderRadius.circular(10),
          borderSide: BorderSide(color: primaryLight.withOpacity(0.5)),
        ),
        focusedBorder: OutlineInputBorder(
          borderRadius: BorderRadius.circular(10),
          borderSide: const BorderSide(color: primaryColor, width: 2),
        ),
        contentPadding: const EdgeInsets.symmetric(horizontal: 16, vertical: 14),
        labelStyle: GoogleFonts.nunito(color: textSecondary),
      ),
      floatingActionButtonTheme: const FloatingActionButtonThemeData(
        backgroundColor: secondaryColor,
        foregroundColor: Colors.white,
        elevation: 4,
      ),
      dividerTheme: DividerThemeData(
        color: primaryLight.withOpacity(0.3),
        thickness: 1,
      ),
      snackBarTheme: SnackBarThemeData(
        behavior: SnackBarBehavior.floating,
        shape: RoundedRectangleBorder(borderRadius: BorderRadius.circular(10)),
      ),
      textSelectionTheme: const TextSelectionThemeData(
        cursorColor: AppTheme.primaryColor,
        selectionColor: Color(0x405C6BC0),
        selectionHandleColor: AppTheme.primaryColor,
      ),
    );
  }
}
```

dart file:<app_drawer.dart>

```dart
// lib/widgets/app_drawer.dart
// ============================================
// MENU LATÉRAL DE L'APPLICATION
// ============================================

import 'package:flutter/material.dart';
import '../services/auth_service.dart';
import '../utils/theme.dart';
import '../screens/home_screen.dart';
import '../screens/task_list_screen.dart';
import '../screens/archive_screen.dart';
import '../screens/report_screen.dart';
import '../screens/calendar_screen.dart';
import '../screens/user_management_screen.dart';
import '../screens/login_screen.dart';

class AppDrawer extends StatelessWidget {
  const AppDrawer({super.key});

  @override
  Widget build(BuildContext context) {
    final auth = AuthService();
    final user = auth.currentUser;

    return Drawer(
      child: Column(
        children: [
          // En-tête du menu
          UserAccountsDrawerHeader(
            decoration: const BoxDecoration(color: AppTheme.primaryColor),
            accountName: Text(
              user?.nomComplet ?? 'Utilisateur',
              style: const TextStyle(fontSize: 18, fontWeight: FontWeight.bold),
            ),
            accountEmail: Text(
              user?.isAdmin == true ? '□ Administrateur' : '□ Exécutant',
              style: const TextStyle(fontSize: 14),
            ),
            currentAccountPicture: CircleAvatar(
              backgroundColor: Colors.white,
              child: Text(
                (user?.prenom.isNotEmpty == true ? user!.prenom[0] : 'U')
                    .toUpperCase(),
                style: const TextStyle(
                  fontSize: 30,
                  fontWeight: FontWeight.bold,
                  color: AppTheme.primaryColor,
                ),
              ),
            ),
          ),

          // Éléments du menu
          _buildDrawerItem(
            context,
            icon: Icons.home,
            title: 'Accueil',
            onTap: () {
              Navigator.pushReplacement(
                context,
                MaterialPageRoute(builder: (_) => const HomeScreen()),
              );
            },
          ),

          _buildDrawerItem(
            context,
            icon: Icons.list_alt,
            title: 'Liste des tâches',
            onTap: () {
              Navigator.pushReplacement(
                context,
                MaterialPageRoute(builder: (_) => const TaskListScreen()),
              );
            },
          ),

          _buildDrawerItem(
            context,
            icon: Icons.calendar_month,
            title: 'Calendrier',
            onTap: () {
              Navigator.pushReplacement(
                context,
                MaterialPageRoute(builder: (_) => const CalendarScreen()),
              );
            },
          ),

          if (auth.isAdmin) ...[
            _buildDrawerItem(
              context,
              icon: Icons.archive,
              title: 'Archives',
              onTap: () {
                Navigator.pushReplacement(
                  context,
```

```
                MaterialPageRoute(builder: (_) => const ArchiveScreen()),
              );
            },
          ),
        ],

        _buildDrawerItem(
          context,
          icon: Icons.assessment,
          title: 'Rapports',
          onTap: () {
            Navigator.pushReplacement(
              context,
              MaterialPageRoute(builder: (_) => const ReportScreen()),
            );
          },
        ),

        if (auth.isAdmin) ...[
          const Divider(),
          _buildDrawerItem(
            context,
            icon: Icons.people,
            title: 'Gestion des utilisateurs',
            onTap: () {
              Navigator.pushReplacement(
                context,
                MaterialPageRoute(
                    builder: (_) => const UserManagementScreen()),
              );
            },
          ),
        ],

        const Spacer(),
        const Divider(),

        _buildDrawerItem(
          context,
          icon: Icons.logout,
          title: 'Déconnexion',
          color: AppTheme.errorColor,
          onTap: () async {
            await auth.logout();
            if (context.mounted) {
              Navigator.pushAndRemoveUntil(
                context,
                MaterialPageRoute(builder: (_) => const LoginScreen()),
                (route) => false,
              );
            }
          },
        ),
        const SizedBox(height: 16),
      ],
    ),
  );
}

Widget _buildDrawerItem(
  BuildContext context, {
  required IconData icon,
  required String title,
  required VoidCallback onTap,
  Color? color,
}) {
  return ListTile(
    leading: Icon(icon, color: color ?? AppTheme.primaryColor),
    title: Text(
      title,
      style: TextStyle(
        fontSize: 16,
        color: color ?? AppTheme.textPrimary,
        fontWeight: FontWeight.w500,
      ),
    ),
    onTap: onTap,
  );
}
}


***dart file:<app_text_field.dart>***

// lib/widgets/app_text_field.dart
// ============================================
// CHAMP TEXTE PERSONNALISÉ
// Place le curseur à l'endroit du tap sans sélectionner
// ============================================
import 'package:flutter/material.dart';

class AppTextField extends StatefulWidget {
  final TextEditingController? controller;
  final String? labelText;
  final String? hintText;
  final String? helperText;
  final Widget? prefixIcon;
  final Widget? suffixIcon;
  final bool obscureText;
  final bool enabled;
  final bool readOnly;
```

```
  final int? maxLines;
  final TextInputType? keyboardType;
  final String? Function(String?)? validator;
  final void Function(String)? onChanged;
  final void Function(String)? onFieldSubmitted;

  const AppTextField({
    super.key,
    this.controller,
    this.labelText,
    this.hintText,
    this.helperText,
    this.prefixIcon,
    this.suffixIcon,
    this.obscureText = false,
    this.enabled = true,
    this.readOnly = false,
    this.maxLines = 1,
    this.keyboardType,
    this.validator,
    this.onChanged,
    this.onFieldSubmitted,
  });

  @override
  State<AppTextField> createState() => _AppTextFieldState();
}

class _AppTextFieldState extends State<AppTextField> {
  late FocusNode _focusNode;
  bool _wasAlreadyFocused = false;

  @override
  void initState() {
    super.initState();
    _focusNode = FocusNode();
    _focusNode.addListener(_onFocusChange);
  }

  @override
  void dispose() {
    _focusNode.removeListener(_onFocusChange);
    _focusNode.dispose();
    super.dispose();
  }

  void _onFocusChange() {
    if (_focusNode.hasFocus) {
      // Le champ vient de recevoir le focus
      // On attend un court instant puis on désélectionne si tout est sélectionné
      Future.delayed(const Duration(milliseconds: 50), () {
        if (widget.controller != null && mounted && _focusNode.hasFocus) {
          final ctrl = widget.controller!;
          final sel = ctrl.selection;
          // Si tout le texte est sélectionné automatiquement par Flutter
          if (sel.baseOffset == 0 &&
              sel.extentOffset == ctrl.text.length &&
              ctrl.text.isNotEmpty &&
              !_wasAlreadyFocused) {
            // Placer le curseur à la fin du texte
            ctrl.selection = TextSelection.collapsed(
              offset: ctrl.text.length,
            );
          }
        }
        _wasAlreadyFocused = true;
      });
    } else {
      // Le champ a perdu le focus
      _wasAlreadyFocused = false;
    }
  }

  @override
  Widget build(BuildContext context) {
    return TextFormField(
      controller: widget.controller,
      focusNode: _focusNode,
      obscureText: widget.obscureText,
      enabled: widget.enabled,
      readOnly: widget.readOnly,
      maxLines: widget.maxLines,
      keyboardType: widget.keyboardType,
      validator: widget.validator,
      onChanged: widget.onChanged,
      onFieldSubmitted: widget.onFieldSubmitted,
      // Désactiver la sélection automatique de tout le texte
      enableInteractiveSelection: true,
      decoration: InputDecoration(
        labelText: widget.labelText,
        hintText: widget.hintText,
        helperText: widget.helperText,
        prefixIcon: widget.prefixIcon,
        suffixIcon: widget.suffixIcon,
        alignLabelWithHint:
            widget.maxLines != null && widget.maxLines! > 1,
      ),
    );
  }
}
```

dart file:<task_card.dart>

```dart
// lib/widgets/task_card.dart
// ============================================
// WIDGET CARTE DE TÂCHE
// ============================================

import 'package:flutter/material.dart';
import 'package:intl/intl.dart';
import '../models/task_model.dart';
import '../utils/theme.dart';

class TaskCard extends StatelessWidget {
  final TaskModel task;
  final VoidCallback? onTap;
  final VoidCallback? onLongPress;

  const TaskCard({
    super.key,
    required this.task,
    this.onTap,
    this.onLongPress,
  });

  @override
  Widget build(BuildContext context) {
    return Card(
      child: InkWell(
        borderRadius: BorderRadius.circular(12),
        onTap: onTap,
        onLongPress: onLongPress,
        child: Padding(
          padding: const EdgeInsets.all(14),
          child: Column(
            crossAxisAlignment: CrossAxisAlignment.start,
            children: [
              // En-tête : numéro + statut
              Row(
                mainAxisAlignment: MainAxisAlignment.spaceBetween,
                children: [
                  // Numéro de tâche + indicateur de synchronisation
                  Row(
                    children: [
                      Text(
                        task.displayNumber,
                        style: const TextStyle(
                          fontSize: 14,
                          fontWeight: FontWeight.bold,
                          color: AppTheme.primaryColor,
                        ),
                      ),
                      if (task.syncStatus != 'synced') ...[
                        const SizedBox(width: 4),
                        Icon(
                          Icons.cloud_off,
                          size: 14,
                          color: AppTheme.warningColor.withValues(alpha: 178),
                        ),
                      ],
                    ],
                  ),
                  _buildStatusChip(),
                ],
              ),
              const SizedBox(height: 8),

              // Immeuble
              Row(
                children: [
                  const Icon(Icons.apartment,
                      size: 18, color: AppTheme.textSecondary),
                  const SizedBox(width: 6),
                  Expanded(
                    child: Text(
                      task.immeuble,
                      style: const TextStyle(
                        fontSize: 16,
                        fontWeight: FontWeight.w600,
                      ),
                    ),
                  ),
                ],
              ),

              // Étage et chambre
              if (task.etage.isNotEmpty || task.chambre.isNotEmpty) ...[
                const SizedBox(height: 4),
                Row(
                  children: [
                    const Icon(Icons.layers,
                        size: 16, color: AppTheme.textSecondary),
                    const SizedBox(width: 6),
                    if (task.etage.isNotEmpty) Text('Étage ${task.etage}'),
                    if (task.etage.isNotEmpty && task.chambre.isNotEmpty)
                      const Text(' — '),
                    if (task.chambre.isNotEmpty) Text('Ch. ${task.chambre}'),
                  ],
                ),
              ],
```

```
            const SizedBox(height: 6),

            // Description
            Text(
              task.description,
              maxLines: 2,
              overflow: TextOverflow.ellipsis,
              style: const TextStyle(
                fontSize: 14,
                color: AppTheme.textSecondary,
              ),
            ),

            const SizedBox(height: 8),

            // Date et exécutant
            Row(
              mainAxisAlignment: MainAxisAlignment.spaceBetween,
              children: [
                // Date de création
                Text(
                  DateFormat('dd/MM/yyyy').format(task.createdAt),
                  style: const TextStyle(
                      fontSize: 12, color: AppTheme.textSecondary),
                ),

                // Date planifiée
                if (task.plannedDate != null)
                  Row(
                    children: [
                      const Icon(Icons.calendar_today,
                          size: 12, color: AppTheme.warningColor),
                      const SizedBox(width: 4),
                      Text(
                        DateFormat('dd/MM/yyyy').format(task.plannedDate!),
                        style: const TextStyle(
                            fontSize: 12, color: AppTheme.warningColor),
                      ),
                    ],
                  ),

                // Exécutant
                if (task.doneBy.isNotEmpty)
                  Row(
                    children: [
                      const Icon(Icons.person,
                          size: 12, color: AppTheme.secondaryColor),
                      const SizedBox(width: 4),
                      Text(
                        task.doneBy,
                        style: const TextStyle(
                            fontSize: 12, color: AppTheme.secondaryColor),
                      ),
                    ],
                  ),
              ],
            ),

            // Date d'exécution si terminée
            if (task.done && task.doneDate != null) ...[
              const SizedBox(height: 4),
              Row(
                children: [
                  const Icon(Icons.event_available,
                      size: 12, color: AppTheme.successColor),
                  const SizedBox(width: 4),
                  Text(
                    'Terminée le ${DateFormat('dd/MM/yyyy').format(task.doneDate!)}',
                    style: const TextStyle(
                        fontSize: 12, color: AppTheme.successColor),
                  ),
                ],
              ),
            ],
          ],
        ),
      ),
    ),
  );
}

Widget _buildStatusChip() {
  Color bgColor;
  Color textColor;
  String label;
  IconData icon;

  if (task.archived) {
    bgColor = AppTheme.archiveColor.withValues(alpha: 51);
    textColor = AppTheme.archiveColor;
    label = 'Archivée';
    icon = Icons.archive;
  } else if (task.done) {
    bgColor = AppTheme.successColor.withValues(alpha: 51);
    textColor = AppTheme.successColor;
    label = 'Terminée';
    icon = Icons.check_circle;
  } else {
    bgColor = AppTheme.warningColor.withValues(alpha: 51);
    textColor = AppTheme.warningColor;
    label = 'En cours';
```

```
      icon = Icons.pending;
    }

    return Container(
      padding: const EdgeInsets.symmetric(horizontal: 10, vertical: 4),
      decoration: BoxDecoration(
        color: bgColor,
        borderRadius: BorderRadius.circular(20),
      ),
      child: Row(
        mainAxisSize: MainAxisSize.min,
        children: [
          Icon(icon, size: 14, color: textColor),
          const SizedBox(width: 4),
          Text(
            label,
            style: TextStyle(
              fontSize: 12,
              color: textColor,
              fontWeight: FontWeight.w600,
            ),
          ),
        ],
      ),
    );
  }
}
```