Dynamic Permission-Based Widget Dashboard System

Flutter + Firebase Architecture Plan

Project: Wall-D Task Management System

Date: December 2025

Status: Architecture Design Document

---

1. Executive Summary

Replace hardcoded screen-based architecture (AdminDesktopScreen, DeveloperDashboardScreen, etc.) with a **dynamic, permission-driven widget system** where:

- Users see **only widgets they have permissions for**
- Each widget represents a discrete feature (create_task, view_assigned_tasks, code_review, etc.)
- Widgets are **arranged dynamically** on a responsive grid/column layout
- Dashboard layout is **personalized per user** based on their permissions
- Admin can configure which widgets users see without code changes

Key Benefits: □ Eliminate hardcoded screens

□ Granular permission control at widget level

□ Scalable to unlimited features

□ Easy to add new widgets without refactoring

□ User-specific dashboard experience

□ Reduced maintenance burden

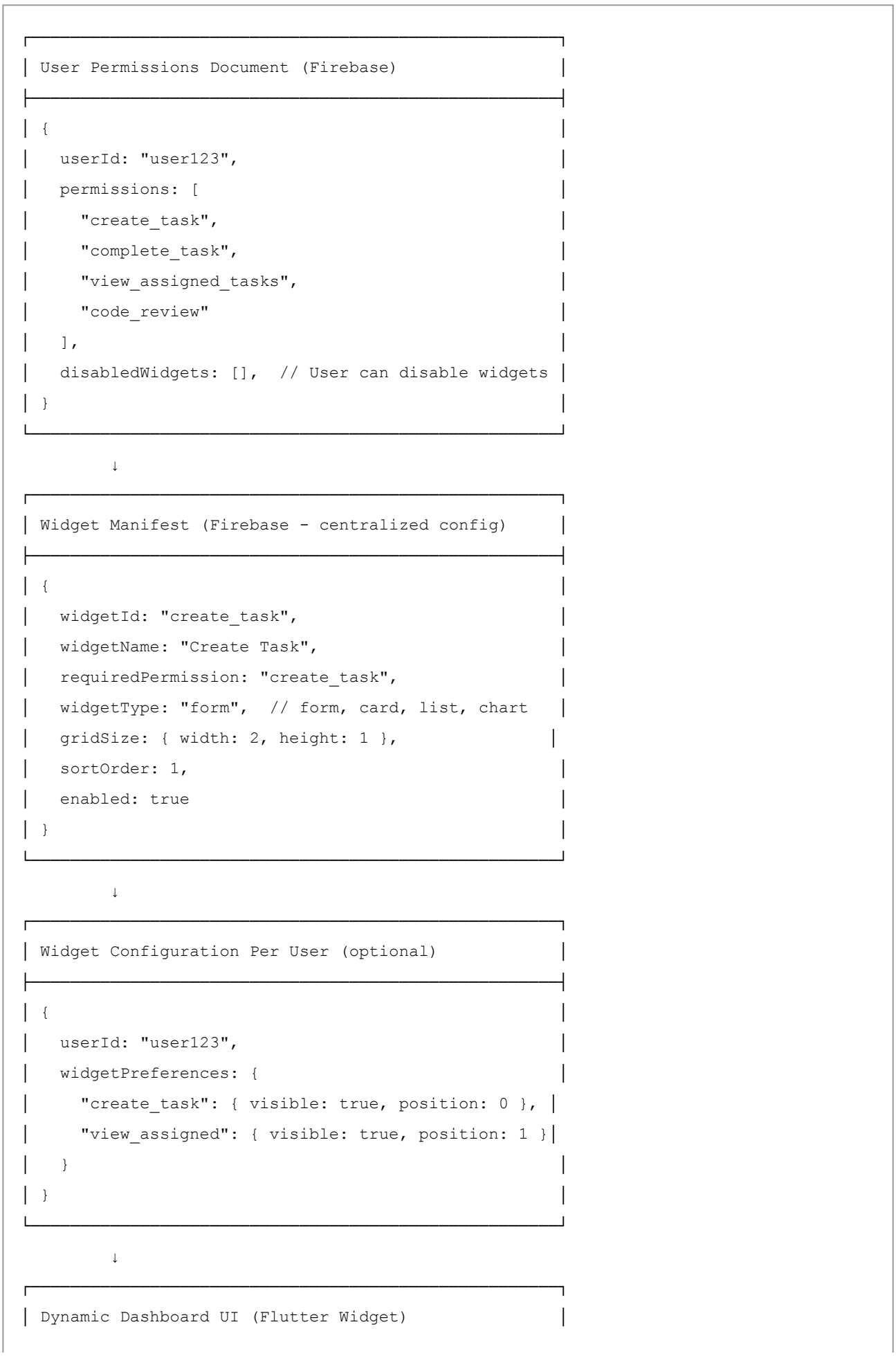
---

2. Architecture Overview

2.1 High-Level Flow Diagram

```
User Login
    ↓
Fetch User Permissions from Firebase
    ↓
Load Widget Manifest (metadata about all available widgets)
    ↓
Filter Widgets by User Permissions
    ↓
Build Dynamic Dashboard with Filtered Widgets
    ↓
Render Widgets on Responsive Grid/Column Layout
    ↓
User sees personalized dashboard
```

2.2 Data Model Stack


User Permissions Document (Firebase)
{
userId: "user123",
permissions: [
"create_task",
"complete_task",
"view_assigned_tasks",
"code_review"
],
disabledWidgets: [], // User can disable widgets
}
↓
Widget Manifest (Firebase - centralized config)
{
widgetId: "create_task",
widgetName: "Create Task",
requiredPermission: "create_task",
widgetType: "form", // form, card, list, chart
gridSize: { width: 2, height: 1 },
sortOrder: 1,
enabled: true
}
↓
Widget Configuration Per User (optional)
{
userId: "user123",
widgetPreferences: {
"create_task": { visible: true, position: 0 },
"view_assigned": { visible: true, position: 1 }
}
}
↓
Dynamic Dashboard UI (Flutter Widget)

```
┌──────────────────────────────────────────────┐
│ Responsive Grid displaying permitted widgets │
└──────────────────────────────────────────────┘
```

---

3. Firebase Firestore Schema Design

3.1 Collections Structure

```
tenants/
  ├── {tenantId}/
  │   ├── users/
  │   │   ├── {userId}/
  │   │   │   ├── profileData: {...}
  │   │   │   ├── permissions: ["create_task", "view_assigned", ...]
  │   │   │   └── role: "admin" | "manager" | "developer" | "employee"
  │   │   │
  │   │
  │   ├── metadata/
  │   │   ├── widgets.json  <-- WIDGET MANIFEST (NEW)
  │   │   ├── designations.json
  │   │   └── rolepermission.json
  │   │
  │   ├── userPreferences/  <-- WIDGET PREFERENCES (NEW)
  │   │   ├── {userId}/
  │   │   │   └── dashboardLayout: {...}
  │   │   │
  │   │
  │   └── tasks/
  │       ├── {taskId}/
  │       │   └── {...}
```

3.2 Widget Manifest Schema (metadata/widgets.json)

```
{
  "widgets": {
    "create_task": {
      "id": "create_task",
      "name": "Create Task",
      "description": "Create new tasks and assign to team members",
      "requiredPermission": "create_task",
      "widgetType": "form",
      "component": "CreateTaskWidget",
      "gridSize": {
        "minWidth": 2,
        "minHeight": 1,
        "defaultWidth": 3,
        "defaultHeight": 2
      },
      "sortOrder": 1,
      "enabled": true,
      "category": "Task Management",
      "icon": "Icons.addtask",
      "refreshInterval": 0,
      "supportsFilters": false,
      "dataSource": "tasks_collection",
      "defaultStyle": {
        "backgroundColor": "Color(0xFF1A1A25)",
        "borderColor": "Color(0x22FFFFFF)"
      }
    },
    "view_assigned_tasks": {
      "id": "view_assigned_tasks",
      "name": "Assigned Tasks",
      "description": "View tasks assigned to you",
      "requiredPermission": "view_assigned_tasks",
      "widgetType": "list",
      "component": "AssignedTasksWidget",
      "gridSize": {
        "minWidth": 2,
        "minHeight": 1,
        "defaultWidth": 3,
        "defaultHeight": 3
      },
      "sortOrder": 2,
      "enabled": true,
      "category": "Task Management",
```

```
    "icon": "Icons.checklist",
    "refreshInterval": 30000,
    "supportsFilters": true,
    "dataSource": "tasks_collection",
    "defaultStyle": {
      "backgroundColor": "Color(0xFF1A1A25)",
      "borderColor": "Color(0x22FFFFFF)"
    }
  },
  "complete_task": {
    "id": "complete_task",
    "name": "Complete Task",
    "description": "Mark tasks as complete",
    "requiredPermission": "complete_task",
    "widgetType": "form",
    "component": "CompleteTaskWidget",
    "gridSize": {
      "minWidth": 2,
      "minHeight": 1,
      "defaultWidth": 3,
      "defaultHeight": 1
    },
    "sortOrder": 3,
    "enabled": true,
    "category": "Task Management",
    "icon": "Icons.checkCircle",
    "refreshInterval": 0,
    "supportsFilters": false,
    "dataSource": "tasks_collection",
    "defaultStyle": {}
  },
  "code_review": {
    "id": "code_review",
    "name": "Code Review",
    "description": "Review code submissions",
    "requiredPermission": "code_review",
    "widgetType": "card",
    "component": "CodeReviewWidget",
    "gridSize": {
      "minWidth": 2,
      "minHeight": 2,
      "defaultWidth": 4,
      "defaultHeight": 2
    },
```

```
    "sortOrder": 4,
    "enabled": true,
    "category": "Development",
    "icon": "Icons.reviewsOutlined",
    "refreshInterval": 60000,
    "supportsFilters": true,
    "dataSource": "codereviews_collection",
    "defaultStyle": {}
  },
  "manage_users": {
    "id": "manage_users",
    "name": "Manage Users",
    "description": "Create, edit, and manage user accounts",
    "requiredPermission": "manage_users",
    "widgetType": "table",
    "component": "ManageUsersWidget",
    "gridSize": {
      "minWidth": 3,
      "minHeight": 2,
      "defaultWidth": 6,
      "defaultHeight": 3
    },
    "sortOrder": 5,
    "enabled": true,
    "category": "Administration",
    "icon": "Icons.peopleManagedOutlined",
    "refreshInterval": 0,
    "supportsFilters": true,
    "dataSource": "users_collection",
    "defaultStyle": {}
  },
  "configure_forms": {
    "id": "configure_forms",
    "name": "Configure Forms",
    "description": "Create and manage dynamic forms",
    "requiredPermission": "configure_forms",
    "widgetType": "form",
    "component": "ConfigureFormsWidget",
    "gridSize": {
      "minWidth": 3,
      "minHeight": 2,
      "defaultWidth": 5,
      "defaultHeight": 3
    },
```

```
    "sortOrder": 6,
    "enabled": true,
    "category": "Administration",
    "icon": "Icons.formsManagedOutlined",
    "refreshInterval": 0,
    "supportsFilters": false,
    "dataSource": "forms_collection",
    "defaultStyle": {}
  },
  "view_all_tasks": {
    "id": "view_all_tasks",
    "name": "View All Tasks",
    "description": "See all tasks in organization",
    "requiredPermission": "view_all_tasks",
    "widgetType": "table",
    "component": "ViewAllTasksWidget",
    "gridSize": {
      "minWidth": 3,
      "minHeight": 2,
      "defaultWidth": 6,
      "defaultHeight": 3
    },
    "sortOrder": 7,
    "enabled": true,
    "category": "Task Management",
    "icon": "Icons.gridViewRounded",
    "refreshInterval": 30000,
    "supportsFilters": true,
    "dataSource": "tasks_collection",
    "defaultStyle": {}
  },
  "approve_task": {
    "id": "approve_task",
    "name": "Approve Tasks",
    "description": "Review and approve task completions",
    "requiredPermission": "approve_task",
    "widgetType": "list",
    "component": "ApproveTaskWidget",
    "gridSize": {
      "minWidth": 2,
      "minHeight": 2,
      "defaultWidth": 3,
      "defaultHeight": 3
    },
```

```
    "sortOrder": 8,
    "enabled": true,
    "category": "Management",
    "icon": "Icons.approvedOutlined",
    "refreshInterval": 30000,
    "supportsFilters": true,
    "dataSource": "approvals_collection",
    "defaultStyle": {}
  },
  "delete_task": {
    "id": "delete_task",
    "name": "Delete Tasks",
    "description": "Delete tasks from system",
    "requiredPermission": "delete_task",
    "widgetType": "form",
    "component": "DeleteTaskWidget",
    "gridSize": {
      "minWidth": 2,
      "minHeight": 1,
      "defaultWidth": 2,
      "defaultHeight": 1
    },
    "sortOrder": 9,
    "enabled": true,
    "category": "Administration",
    "icon": "Icons.deleteSweep",
    "refreshInterval": 0,
    "supportsFilters": false,
    "dataSource": "tasks_collection",
    "defaultStyle": {}
  },
  "export_data": {
    "id": "export_data",
    "name": "Export Data",
    "description": "Export tasks and reports to CSV/Excel",
    "requiredPermission": "export_data",
    "widgetType": "form",
    "component": "ExportDataWidget",
    "gridSize": {
      "minWidth": 2,
      "minHeight": 1,
      "defaultWidth": 2,
      "defaultHeight": 1
    },
```

```
      "sortOrder": 10,
      "enabled": true,
      "category": "Administration",
      "icon": "Icons.downloadForOffline",
      "refreshInterval": 0,
      "supportsFilters": false,
      "dataSource": "multiple",
      "defaultStyle": {}
    }
  }
}
```

3.3 User Permissions Document (users/

```
{
  "userId": "user123",
  "email": "developer@wall-d.com",
  "profileData": {
    "fullName": "John Developer",
    "avatar": "..."
  },
  "permissions": [
    "create_task",
    "complete_task",
    "view_assigned_tasks",
    "code_review"
  ],
  "role": "developer",
  "designation": "Senior Developer",
  "department": "Engineering",
  "status": "active",
  "createdAt": "timestamp"
}
```

3.4 User Dashboard Preferences (userPreferences/

```
{
  "userId": "user123",
  "dashboardLayout": {
    "widgetOrder": [
      "view_assigned_tasks",
      "create_task",
      "code_review",
      "complete_task"
    ],
    "widgetSettings": {
      "view_assigned_tasks": {
        "visible": true,
        "position": 0,
        "width": 3,
        "height": 3,
        "refreshInterval": 30000
      },
      "create_task": {
        "visible": true,
        "position": 1,
        "width": 3,
        "height": 2
      },
      "code_review": {
        "visible": true,
        "position": 2,
        "width": 4,
        "height": 2
      },
      "complete_task": {
        "visible": true,
        "position": 3,
        "width": 3,
        "height": 1
      }
    },
    "theme": "dark",
    "gridColumns": 6,
    "lastUpdated": "timestamp"
  }
}
```

4. Dart/Flutter Data Models

4.1 Widget Model Classes

File: `lib/models/widget_models.dart`

```dart
/// Represents a widget configuration from manifest
class WidgetConfig {
  final String id;
  final String name;
  final String description;
  final String requiredPermission;
  final String widgetType; // form, card, list, table, chart
  final String component; // Flutter widget class name
  final GridSize gridSize;
  final int sortOrder;
  final bool enabled;
  final String category;
  final String icon;
  final int refreshInterval; // milliseconds, 0 = no refresh
  final bool supportsFilters;
  final String dataSource;
  final Map<String, dynamic> defaultStyle;

  WidgetConfig({
    required this.id,
    required this.name,
    required this.description,
    required this.requiredPermission,
    required this.widgetType,
    required this.component,
    required this.gridSize,
    required this.sortOrder,
    required this.enabled,
    required this.category,
    required this.icon,
    required this.refreshInterval,
    required this.supportsFilters,
    required this.dataSource,
    required this.defaultStyle,
  });

  factory WidgetConfig.fromJson(String id, Map<String, dynamic> json) {
    return WidgetConfig(
      id: id,
      name: json['name'] ?? '',
      description: json['description'] ?? '',
      requiredPermission: json['requiredPermission'] ?? '',
      widgetType: json['widgetType'] ?? 'card',
```

```
      component: json['component'] ?? '',
      gridSize: GridSize.fromJson(json['gridSize'] ?? {}),
      sortOrder: json['sortOrder'] ?? 99,
      enabled: json['enabled'] ?? true,
      category: json['category'] ?? 'Other',
      icon: json['icon'] ?? 'Icons.widget',
      refreshInterval: json['refreshInterval'] ?? 0,
      supportsFilters: json['supportsFilters'] ?? false,
      dataSource: json['dataSource'] ?? '',
      defaultStyle: json['defaultStyle'] ?? {},
    );
  }
}

/// Grid sizing configuration for widget
class GridSize {
  final int minWidth;
  final int minHeight;
  final int defaultWidth;
  final int defaultHeight;

  GridSize({
    required this.minWidth,
    required this.minHeight,
    required this.defaultWidth,
    required this.defaultHeight,
  });

  factory GridSize.fromJson(Map<String, dynamic> json) {
    return GridSize(
      minWidth: json['minWidth'] ?? 1,
      minHeight: json['minHeight'] ?? 1,
      defaultWidth: json['defaultWidth'] ?? 2,
      defaultHeight: json['defaultHeight'] ?? 1,
    );
  }
}

/// User dashboard layout preferences
class DashboardLayout {
  final String userId;
  final List<String> widgetOrder;
  final Map<String, WidgetPreference> widgetSettings;
  final String theme;
```

```
  final int gridColumns;
  final DateTime lastUpdated;

  DashboardLayout({
    required this.userId,
    required this.widgetOrder,
    required this.widgetSettings,
    required this.theme,
    required this.gridColumns,
    required this.lastUpdated,
  });

  factory DashboardLayout.fromJson(String userId, Map<String, dynamic> json) {
    return DashboardLayout(
      userId: userId,
      widgetOrder: List<String>.from(json['widgetOrder'] ?? []),
      widgetSettings: (json['widgetSettings'] as Map<String, dynamic>?)
              ?.map(
            (key, value) => MapEntry(
              key,
              WidgetPreference.fromJson(value as Map<String, dynamic>),
            ),
          ) ??
          {},
      theme: json['theme'] ?? 'dark',
      gridColumns: json['gridColumns'] ?? 6,
      lastUpdated: (json['lastUpdated'] as Timestamp?)?.toDate() ?? DateTime.now(),
    );
  }
}

/// Per-widget user preferences
class WidgetPreference {
  final bool visible;
  final int position;
  final int width;
  final int height;
  final int? refreshInterval;
  final Map<String, dynamic>? customSettings;

  WidgetPreference({
    required this.visible,
    required this.position,
    required this.width,
```

```
    required this.height,
    this.refreshInterval,
    this.customSettings,
  });

  factory WidgetPreference.fromJson(Map<String, dynamic> json) {
    return WidgetPreference(
      visible: json['visible'] ?? true,
      position: json['position'] ?? 0,
      width: json['width'] ?? 2,
      height: json['height'] ?? 1,
      refreshInterval: json['refreshInterval'],
      customSettings: json['customSettings'],
    );
  }
}

/// User with permissions
class UserWithPermissions {
  final String userId;
  final String email;
  final String fullName;
  final List<String> permissions;
  final String role;
  final String designation;
  final DateTime createdAt;

  UserWithPermissions({
    required this.userId,
    required this.email,
    required this.fullName,
    required this.permissions,
    required this.role,
    required this.designation,
    required this.createdAt,
  });

  bool hasPermission(String permission) => permissions.contains(permission);
}
```

---

5. Repository Layer (Firebase Integration)

5.1 Widget Repository

File: `lib/repositories/widget_repository.dart`

```dart
import 'package:cloud_firestore/cloud_firestore.dart';
import '../models/widget_models.dart';

class WidgetRepository {
  final FirebaseFirestore _db;

  WidgetRepository(this._db);

  /// Load all available widgets from manifest
  Future<Map<String, WidgetConfig>> loadWidgetManifest(String tenantId) async {
    try {
      final doc = await _db
          .collection('tenants')
          .doc(tenantId)
          .collection('metadata')
          .doc('widgets.json')
          .get();

      if (!doc.exists) {
        print('Widget manifest not found for tenant: $tenantId');
        return {};
      }

      final data = doc.data() as Map<String, dynamic>;
      final widgets = data['widgets'] as Map<String, dynamic>? ?? {};

      return widgets.map(
        (id, config) => MapEntry(
          id,
          WidgetConfig.fromJson(id, config as Map<String, dynamic>),
        ),
      );
    } catch (e) {
      print('Error loading widget manifest: $e');
      return {};
    }
  }

  /// Get user permissions from Firestore
  Future<List<String>> getUserPermissions(
    String tenantId,
    String userId,
  ) async {
```

```
  try {
    final doc = await _db
        .collection('tenants')
        .doc(tenantId)
        .collection('users')
        .doc(userId)
        .get();

    if (!doc.exists) return [];

    final data = doc.data() as Map<String, dynamic>;
    return List<String>.from(data['permissions'] ?? []);
  } catch (e) {
    print('Error loading user permissions: $e');
    return [];
  }
}

/// Get enabled widgets for user (filtered by permissions)
Future<List<WidgetConfig>> getEnabledWidgetsForUser(
  String tenantId,
  String userId,
) async {
  final manifest = await loadWidgetManifest(tenantId);
  final permissions = await getUserPermissions(tenantId, userId);

  // Filter widgets by permissions and enabled status
  final widgets = manifest.values
      .where(
        (w) =>
            w.enabled &&
            (w.requiredPermission.isEmpty ||
                permissions.contains(w.requiredPermission)),
      )
      .toList();

  // Sort by sortOrder
  widgets.sort((a, b) => a.sortOrder.compareTo(b.sortOrder));

  return widgets;
}

/// Save user dashboard layout preferences
Future<void> saveDashboardLayout(
```

```
  String tenantId,
  String userId,
  DashboardLayout layout,
) async {
  try {
    await _db
        .collection('tenants')
        .doc(tenantId)
        .collection('userPreferences')
        .doc(userId)
        .set(
      {
        'userId': userId,
        'dashboardLayout': {
          'widgetOrder': layout.widgetOrder,
          'widgetSettings': layout.widgetSettings.map(
            (key, pref) => MapEntry(key, {
              'visible': pref.visible,
              'position': pref.position,
              'width': pref.width,
              'height': pref.height,
              'refreshInterval': pref.refreshInterval,
            }),
          ),
          'theme': layout.theme,
          'gridColumns': layout.gridColumns,
          'lastUpdated': FieldValue.serverTimestamp(),
        },
      },
    );
  } catch (e) {
    print('Error saving dashboard layout: $e');
  }
}

/// Load user dashboard layout preferences
Future<DashboardLayout?> loadDashboardLayout(
  String tenantId,
  String userId,
) async {
  try {
    final doc = await _db
        .collection('tenants')
        .doc(tenantId)
```

```
          .collection('userPreferences')
          .doc(userId)
          .get();

      if (!doc.exists) return null;

      final data = doc.data() as Map<String, dynamic>;
      final layoutData = data['dashboardLayout'] as Map<String, dynamic>?;

      if (layoutData == null) return null;

      return DashboardLayout.fromJson(userId, layoutData);
    } catch (e) {
      print('Error loading dashboard layout: $e');
      return null;
    }
  }
}
```

---

6. Widget Factory & Resolver

6.1 Dynamic Widget Builder

File: `lib/services/widget_factory.dart`

```dart
import 'package:flutter/material.dart';
import '../models/widget_models.dart';
import '../widgets/dashboard_widgets/index.dart'; // All widget implementations

/// Factory to instantiate widget components dynamically
class WidgetFactory {
  /// Build a widget based on WidgetConfig
  static Widget buildWidget({
    required WidgetConfig config,
    required String tenantId,
    required String userId,
    required VoidCallback onRefresh,
  }) {
    switch (config.component) {
      // Task Management Widgets
      case 'CreateTaskWidget':
        return CreateTaskWidget(
          tenantId: tenantId,
          userId: userId,
          widgetConfig: config,
        );

      case 'AssignedTasksWidget':
        return AssignedTasksWidget(
          tenantId: tenantId,
          userId: userId,
          widgetConfig: config,
        );

      case 'CompleteTaskWidget':
        return CompleteTaskWidget(
          tenantId: tenantId,
          userId: userId,
          widgetConfig: config,
        );

      case 'ViewAllTasksWidget':
        return ViewAllTasksWidget(
          tenantId: tenantId,
          userId: userId,
          widgetConfig: config,
        );
```

```
      case 'ApproveTaskWidget':
        return ApproveTaskWidget(
          tenantId: tenantId,
          userId: userId,
          widgetConfig: config,
        );

      case 'DeleteTaskWidget':
        return DeleteTaskWidget(
          tenantId: tenantId,
          userId: userId,
          widgetConfig: config,
        );

      // Code Review Widget
      case 'CodeReviewWidget':
        return CodeReviewWidget(
          tenantId: tenantId,
          userId: userId,
          widgetConfig: config,
        );

      // Admin Widgets
      case 'ManageUsersWidget':
        return ManageUsersWidget(
          tenantId: tenantId,
          userId: userId,
          widgetConfig: config,
        );

      case 'ConfigureFormsWidget':
        return ConfigureFormsWidget(
          tenantId: tenantId,
          userId: userId,
          widgetConfig: config,
        );

      case 'ExportDataWidget':
        return ExportDataWidget(
          tenantId: tenantId,
          userId: userId,
          widgetConfig: config,
        );
```

```
      // Fallback
      default:
        return _buildErrorWidget(config);
    }
  }

  static Widget _buildErrorWidget(WidgetConfig config) {
    return Card(
      child: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: [
            const Icon(Icons.error, size: 48, color: Colors.red),
            const SizedBox(height: 8),
            Text('Unknown widget: ${config.component}'),
          ],
        ),
      ),
    );
  }
}

/// Get icon from string identifier
IconData getIconFromString(String iconString) {
  // Parse Icons.xyz format
  final iconName = iconString.replaceAll('Icons.', '');

  // Map common icon names
  const iconMap = {
    'addtask': Icons.addtask,
    'checklist': Icons.checklist,
    'checkCircle': Icons.checkCircle,
    'reviewsOutlined': Icons.reviewsOutlined,
    'peopleManagedOutlined': Icons.peopleManagedOutlined,
    'formsManagedOutlined': Icons.formsManagedOutlined,
    'gridViewRounded': Icons.gridViewRounded,
    'approvedOutlined': Icons.approvedOutlined,
    'deleteSweep': Icons.deleteSweep,
    'downloadForOffline': Icons.downloadForOffline,
  };

  return iconMap[iconName] ?? Icons.widget;
}
```

7. Dynamic Dashboard Screen

7.1 Main Dashboard Widget

File: `lib/screens/dynamic_dashboard_screen.dart`

```dart
import 'package:flutter/material.dart';
import '../models/widget_models.dart';
import '../repositories/widget_repository.dart';
import '../services/widget_factory.dart';

class DynamicDashboardScreen extends StatefulWidget {
  final String tenantId;
  final String userId;

  const DynamicDashboardScreen({
    Key? key,
    required this.tenantId,
    required this.userId,
  }) : super(key: key);

  @override
  State<DynamicDashboardScreen> createState() => _DynamicDashboardScreenState();
}

class _DynamicDashboardScreenState extends State<DynamicDashboardScreen> {
  late WidgetRepository _widgetRepo;
  List<WidgetConfig> _enabledWidgets = [];
  DashboardLayout? _userLayout;
  bool _loading = true;
  String? _error;

  @override
  void initState() {
    super.initState();
    _widgetRepo = WidgetRepository(FirebaseFirestore.instance);
    _loadDashboard();
  }

  Future<void> _loadDashboard() async {
    try {
      setState(() => _loading = true);

      // Load enabled widgets for user
      _enabledWidgets = await _widgetRepo.getEnabledWidgetsForUser(
        widget.tenantId,
        widget.userId,
      );
```

```
    // Load user's layout preferences (or create default)
    _userLayout = await _widgetRepo.loadDashboardLayout(
      widget.tenantId,
      widget.userId,
    );

    if (_userLayout == null) {
      _userLayout = _createDefaultLayout(_enabledWidgets);
    }

    setState(() => _loading = false);
  } catch (e) {
    setState(() {
      _error = e.toString();
      _loading = false;
    });
  }
}

DashboardLayout _createDefaultLayout(List<WidgetConfig> widgets) {
  final widgetOrder = widgets.map((w) => w.id).toList();
  final widgetSettings = {
    for (var i = 0; i < widgets.length; i++)
      widgets[i].id: WidgetPreference(
        visible: true,
        position: i,
        width: widgets[i].gridSize.defaultWidth,
        height: widgets[i].gridSize.defaultHeight,
      ),
  };

  return DashboardLayout(
    userId: widget.userId,
    widgetOrder: widgetOrder,
    widgetSettings: widgetSettings,
    theme: 'dark',
    gridColumns: 6,
    lastUpdated: DateTime.now(),
  );
}

@override
Widget build(BuildContext context) {
  if (_loading) {
```

```
    return const Scaffold(
      body: Center(
        child: CircularProgressIndicator(),
      ),
    );
  }

  if (_error != null) {
    return Scaffold(
      body: Center(
        child: Text('Error: $_error'),
      ),
    );
  }

  if (_enabledWidgets.isEmpty) {
    return const Scaffold(
      body: Center(
        child: Text('No widgets available for this user'),
      ),
    );
  }

  return Scaffold(
    appBar: AppBar(
      title: const Text('Dashboard'),
      actions: [
        IconButton(
          icon: const Icon(Icons.refresh),
          onPressed: _loadDashboard,
        ),
        IconButton(
          icon: const Icon(Icons.settings),
          onPressed: _showLayoutSettings,
        ),
      ],
    ),
    body: _buildDashboardGrid(),
  );
}

/// Build responsive grid of widgets
Widget _buildDashboardGrid() {
  // Get visible widgets in order
```

```
  final visibleWidgets = _userLayout!.widgetOrder
      .where((widgetId) =>
          _userLayout!.widgetSettings[widgetId]?.visible ?? false)
      .map((widgetId) =>
          _enabledWidgets.firstWhere((w) => w.id == widgetId))
      .toList();

  // Mobile: Single column layout
  if (MediaQuery.of(context).size.width < 900) {
    return ListView.builder(
      padding: const EdgeInsets.all(16),
      itemCount: visibleWidgets.length,
      itemBuilder: (context, index) {
        final widget = visibleWidgets[index];
        final pref = _userLayout!.widgetSettings[widget.id];

        return Padding(
          padding: const EdgeInsets.only(bottom: 16),
          child: _buildWidgetContainer(widget, pref),
        );
      },
    );
  }

  // Desktop: Responsive grid
  return SingleChildScrollView(
    padding: const EdgeInsets.all(16),
    child: Wrap(
      spacing: 16,
      runSpacing: 16,
      children: visibleWidgets.map((widget) {
        final pref = _userLayout!.widgetSettings[widget.id];
        return SizedBox(
          width: _calculateWidgetWidth(pref!.width),
          child: _buildWidgetContainer(widget, pref),
        );
      }).toList(),
    ),
  );
}

/// Calculate widget width based on grid columns
double _calculateWidgetWidth(int gridSpan) {
  final screenWidth = MediaQuery.of(context).size.width - 32; // padding
```

```
  return (screenWidth / _userLayout!.gridColumns) * gridSpan;
}

/// Build individual widget with error handling
Widget _buildWidgetContainer(
  WidgetConfig config,
  WidgetPreference? pref,
) {
  return Container(
    decoration: BoxDecoration(
      color: const Color(0xFF1A1A25),
      borderRadius: BorderRadius.circular(12),
      border: Border.all(color: const Color(0x22FFFFFF)),
    ),
    padding: const EdgeInsets.all(12),
    child: Column(
      crossAxisAlignment: CrossAxisAlignment.start,
      children: [
        // Widget header with title and icon
        Row(
          children: [
            Icon(
              getIconFromString(config.icon),
              color: Colors.cyan,
              size: 20,
            ),
            const SizedBox(width: 8),
            Expanded(
              child: Text(
                config.name,
                style: const TextStyle(
                  color: Colors.white,
                  fontSize: 14,
                  fontWeight: FontWeight.w600,
                ),
              ),
            ),
            // More options menu
            PopupMenuButton(
              itemBuilder: (context) => [
                PopupMenuItem(
                  child: const Text('Hide'),
                  onTap: () => _hideWidget(config.id),
                ),
```

```
                PopupMenuItem(
                  child: const Text('Settings'),
                  onTap: () => _showWidgetSettings(config),
                ),
              ],
            ),
          ],
        ),
        const SizedBox(height: 12),
        // Widget content
        Expanded(
          child: WidgetFactory.buildWidget(
            config: config,
            tenantId: widget.tenantId,
            userId: widget.userId,
            onRefresh: () => setState(() {}),
          ),
        ),
      ],
    ),
  );
}

void _hideWidget(String widgetId) {
  setState(() {
    if (_userLayout!.widgetSettings[widgetId] != null) {
      final oldPref = _userLayout!.widgetSettings[widgetId]!;
      _userLayout!.widgetSettings[widgetId] = WidgetPreference(
        visible: false,
        position: oldPref.position,
        width: oldPref.width,
        height: oldPref.height,
      );
    }
  });
  _saveDashboardLayout();
}

void _showLayoutSettings() {
  // TODO: Implement layout customization dialog
}

void _showWidgetSettings(WidgetConfig config) {
  // TODO: Implement widget-specific settings
```

```
  }

  void _saveDashboardLayout() {
    _widgetRepo.saveDashboardLayout(
      widget.tenantId,
      widget.userId,
      _userLayout!,
    );
  }
}

/// Helper to parse icon from string
IconData getIconFromString(String iconString) {
  return WidgetFactory_getIconFromString(iconString);
}
```

---

8. Implementation Path (Phased Approach)

Phase 1: Data Layer (Week 1)

- [ ] Create widget manifest schema in Firebase (metadata/widgets.json)
- [ ] Create model classes (WidgetConfig, DashboardLayout, etc.)
- [ ] Create WidgetRepository with CRUD operations
- [ ] Add sample widget manifest data for testing

Deliverables:

- Widget models working with Firebase Firestore
- Repository methods tested
- Initial widget manifest populated

Phase 2: Widget Infrastructure (Week 2)

- [ ] Create abstract base widget class for all dashboard widgets
- [ ] Implement WidgetFactory for dynamic instantiation
- [ ] Create reusable widget container with loading/error states
- [ ] Implement permission checking utilities

Deliverables:

- Factory pattern working

- Base widget structure established
- Widget container with consistent styling

Phase 3: Dashboard Screen (Week 2-3)

- [ ] Build DynamicDashboardScreen with responsive layout
- [ ] Implement grid/column layout system
- [ ] Add widget visibility toggling
- [ ] Implement default layout creation

Deliverables:

- Functional dashboard screen
- Widgets displaying correctly
- Layout switching (desktop/mobile) working

Phase 4: Individual Widgets (Week 3-4)

Implement each widget class:

- [ ] CreateTaskWidget
- [ ] AssignedTasksWidget
- [ ] CompleteTaskWidget
- [ ] CodeReviewWidget
- [ ] ViewAllTasksWidget
- [ ] ApproveTaskWidget
- [ ] ManageUsersWidget
- [ ] ConfigureFormsWidget
- [ ] DeleteTaskWidget
- [ ] ExportDataWidget

Deliverables:

- All 10+ widgets implemented
- Each widget functional and tested

Phase 5: User Preferences (Week 4)

- [ ] Implement layout customization UI
- [ ] Add drag-and-drop reordering (optional)
- [ ] Save/load user preferences from Firestore
- [ ] Add theme switching

Deliverables:

- Users can customize their dashboard

- Preferences persisted to Firebase
- Clean UX for customization

Phase 6: Testing & Optimization (Week 5)

- [ ] Test permission filtering
- [ ] Performance optimize widget loading
- [ ] Add real-time updates with Firestore listeners
- [ ] Error handling and fallbacks

Deliverables:

- All features tested
- Performance optimized
- Production-ready

---

9. Key Implementation Details

9.1 Permission Checking Pattern

```
// In widget build method
if (!widget.userPermissions.contains(widget.widgetConfig.requiredPermission)) {
  return Container(
    color: Colors.red.withOpacity(0.1),
    child: const Center(
      child: Text('You do not have permission to access this widget'),
    ),
  );
}
```

9.2 Widget Lifecycle

```
abstract class DashboardWidget extends StatefulWidget {
  final WidgetConfig widgetConfig;
  final String tenantId;
  final String userId;
  final List<String> userPermissions;

  const DashboardWidget({
    required this.widgetConfig,
    required this.tenantId,
    required this.userId,
    required this.userPermissions,
  });
}

abstract class DashboardWidgetState<T extends DashboardWidget>
    extends State<T> {
  /// Override to load widget data
  Future<void> loadData();

  /// Override to refresh widget data
  Future<void> refreshData();

  /// Check if user has required permission
  bool hasPermission(String permission) {
    return widget.userPermissions.contains(permission);
  }
}
```

9.3 Real-time Updates

```
/// Add Firestore listener for real-time updates
StreamBuilder(
  stream: _db
      .collection('tenants')
      .doc(widget.tenantId)
      .collection(widget.widgetConfig.dataSource)
      .where('assignedTo', isEqualTo: widget.userId)
      .orderBy('createdAt', descending: true)
      .snapshots(),
  builder: (context, snapshot) {
    if (snapshot.hasError) {
      return Center(child: Text('Error: ${snapshot.error}'));
    }
    if (!snapshot.hasData) {
      return const Center(child: CircularProgressIndicator());
    }

    final docs = snapshot.data!.docs;
    return ListView.builder(
      itemCount: docs.length,
      itemBuilder: (context, index) {
        // Build list item from doc
      },
    );
  },
)
```

---

10. Navigation from Auth Screen

10.1 Updated Auth Flow

```
// In AuthScreen - after user login
Future<void> _navigateToPersonalizedDashboard(String userId) async {
  final userDoc = await _firestore
      .collection('tenants')
      .doc(widget.tenantId)
      .collection('users')
      .doc(userId)
      .get();

  if (!userDoc.exists) {
    ScaffoldMessenger.of(context).showSnackBar(
      const SnackBar(content: Text('User data not found')),
    );
    return;
  }

  // Instead of routing to hardcoded screens
  // Route directly to DynamicDashboardScreen
  Navigator.of(context).pushReplacementNamed(
    '/dashboard',
    arguments: {
      'tenantId': widget.tenantId,
      'userId': userId,
    },
  );
}
```

---

11. Permission Configuration Examples

Example 1: Developer User

```
{
  "userId": "dev_user_123",
  "permissions": [
    "create_task",
    "complete_task",
    "view_assigned_tasks",
    "code_review"
  ],
  "role": "developer"
}
```

Dashboard shows:

1. Assigned Tasks (view_assigned_tasks)
2. Create Task (create_task)
3. Code Review (code_review)
4. Complete Task (complete_task)

Example 2: Manager User

```
{
  "userId": "manager_user_456",
  "permissions": [
    "manage_users",
    "configure_forms",
    "view_all_tasks",
    "approve_task",
    "delete_task",
    "export_data"
  ],
  "role": "manager"
}
```

Dashboard shows:

1. View All Tasks (view_all_tasks)
2. Approve Tasks (approve_task)
3. Manage Users (manage_users)
4. Configure Forms (configure_forms)
5. Export Data (export_data)
6. Delete Tasks (delete_task)

Example 3: Admin User

```
{
  "userId": "admin_user_789",
  "permissions": [
    "create_task",
    "complete_task",
    "view_assigned_tasks",
    "code_review",
    "manage_users",
    "configure_forms",
    "view_all_tasks",
    "approve_task",
    "delete_task",
    "export_data"
  ],
  "role": "admin"
}
```

Dashboard shows: All 10+ widgets

---

12. Benefits of This Architecture

 Scalability - Add new widgets without changing screen code

 Flexibility - Permissions can be changed in Firebase without app updates

 Maintainability - No hardcoded screens to manage

 Reusability - Each widget is self-contained and modular

 User Experience - Users see only features they need

 Dynamic Configuration - Change widget manifest for instant updates

 Real-time - Firestore listeners keep dashboards fresh

 Mobile Support - Responsive design works on all screen sizes

---

13. Future Enhancements

1. **Drag-and-drop reordering** - Let users arrange widgets
2. **Widget resizing** - Dynamic widget dimension adjustment
3. **Custom themes** - Per-widget and dashboard-wide theming
4. **Export/import layouts** - Share dashboard configs
5. **Role-based defaults** - Automatic layouts per role
6. **Widget caching** - Performance optimization
7. **Analytics** - Track widget usage
8. **Notifications** - Widget-level alerts

---

14. Quick Start Checklist

- [ ] Create `lib/models/widget_models.dart`
- [ ] Create `lib/repositories/widget_repository.dart`
- [ ] Create `lib/services/widget_factory.dart`
- [ ] Create `lib/screens/dynamic_dashboard_screen.dart`
- [ ] Add widget manifest to Firebase
- [ ] Update permissions structure in user documents
- [ ] Create widget implementations directory
- [ ] Update auth flow to route to new dashboard
- [ ] Remove hardcoded AdminDesktopScreen, DeveloperDashboardScreen, etc.
- [ ] Test with sample permissions

Conclusion

This dynamic widget architecture replaces hardcoded screens with a flexible, permission-driven system that scales with your application. Users see only what they're authorized to use, and admins can manage access through Firebase without code changes.

The system is production-ready and supports real-time updates, responsive design, and unlimited widget expansion.