```dart
import 'dart:async';
import 'package:flutter/foundation.dart';
import 'package:flutter/scheduler.dart';
import 'package:flutter/material.dart' as flutter;
import '../models/log_entry.dart';
import '../views/debug_wrapper.dart';
import '../models/log_config.dart';
import '../models/log_navigation_callbacks.dart';
import '../views/default_error_widget.dart';

/// Pure Flutter log manager using ChangeNotifier for state management
class SuperLogManager extends ChangeNotifier {
 static SuperLogManager? _instance;
 static SuperLogConfig? _config;
 static bool _isCapturingPrint = false;
 static bool _isInDebugPrint = false;
 static final flutter.GlobalKey<flutter.NavigatorState> navigatorKey =
    flutter.GlobalKey<flutter.NavigatorState>(
  debugLabel: 'SuperLogManagerNavigator',
 );

 static SuperLogManager init({SuperLogConfig? config}) {
  _config = config ?? const SuperLogConfig();
  if (!_config!.enabled) {
    return _instance ??= SuperLogManager._disabled();
```

```
  }
  _instance ??= SuperLogManager._();
  return _instance!;
}

static bool get isInitialized =>
    _instance != null && _config?.enabled != false;

static SuperLogConfig? get config => _config;

static SuperLogNavigationCallbacks get navigationCallbacks {
  final configCallbacks = _config?.navigationCallbacks;
  // If no callbacks provided, use default Flutter Navigator callbacks
  // MaterialApp.builder provides context with Navigator access
  // For GetX and go_router, users should provide callbacks if needed
  return configCallbacks ?? SuperLogNavigationCallbacks.flutter();
}

static SuperLogManager get instance {
  if (_config?.enabled == false) {
    return _instance ??= SuperLogManager._disabled();
  }
  _instance ??= SuperLogManager._();
  return _instance!;
}
```

```
/// Run app with automatic error handling
///
/// Usage:
/// ```dart
/// SuperLogManager.runApp(
///   const MyApp(),
///   config: const SuperLogConfig(
///     enabled: true,
///     showOverlayBubble: true,
///   ),
/// );
/// ```
static void runApp(
  flutter.Widget app, {
  SuperLogConfig? config,
  flutter.Widget Function(Object error)? errorWidget,
  FutureOr<bool> Function()? preRun,
  void Function()? postRun,
}) async {
  final finalConfig = config ?? const SuperLogConfig();
  final finalErrorWidget =
      errorWidget ?? ((error) => SuperDefaultErrorWidget(error));

  // Initialize bindings first (in current zone)
  flutter.WidgetsFlutterBinding.ensureInitialized();
```

```
// Initialize LogManager if enabled
if (finalConfig.enabled) {
  init(config: finalConfig);
}

// Set up error handlers (works in same zone)
if (finalConfig.enabled) {
  _setupErrorHandlers(finalConfig, finalErrorWidget);
}

try {
  // Run preRun hook
  if (preRun != null) {
    final preRunResult = await preRun();
    if (preRunResult == false) {
      debugPrint('preRun returned false, aborting app start');
      return;
    }
  }

  // Run app - wrap with debug support if enabled
  flutter.runApp(
    finalConfig.enabled && finalConfig.showOverlayBubble
        ? SuperDebugWrapper(child: app)
        : app,
  );
```

```
  // Run postRun hook after first frame
  if (postRun != null) {
    SchedulerBinding.instance.addPostFrameCallback((_) {
      Future.microtask(() async {
        try {
          postRun();
        } catch (e) {
          debugPrint('Error in postRun: $e');
          if (isInitialized) {
            try {
              instance.addLog(
                'Error in postRun: $e',
                level: LogLevel.error,
                error: e,
              );
            } catch (_) {}
          }
        }
      });
    });
  }
} catch (e, s) {
  debugPrint('Error in runApp: $e\n$s');
  if (finalConfig.enabled && isInitialized) {
    try {
```

```
        instance.addLog(
          'Error in runApp: $e',
          level: LogLevel.error,
          error: e,
          stackTrace: s,
        );
      } catch (_) {}
    }
    flutter.runApp(finalErrorWidget(e));
  }
}

/// Wrap app with debug support
/// Detects MaterialApp and injects builder

static void _setupErrorHandlers(
  SuperLogConfig config,
  flutter.Widget Function(Object error) errorWidget,
) {
  // Handle Flutter framework errors
  flutter.FlutterError.onError = (flutter.FlutterErrorDetails details) {
    if (isInitialized) {
      try {
        instance.addLog(
          details.exceptionAsString(),
          level: LogLevel.error,
```

```
      error: details.exception,
      stackTrace: details.stack,
    );
   } catch (_) {}
  }
};

// Handle Dart errors
PlatformDispatcher.instance.onError = (error, stack) {
  if (isInitialized) {
   try {
    instance.addLog(
      error.toString(),
      level: LogLevel.error,
      error: error,
      stackTrace: stack,
    );
   } catch (_) {}
  }
  return false;
};

// Set up print and debugPrint interception
if (config.capturePrint || config.captureDebugPrint) {
  _hookDebugPrint(config);
}
```

```
}

static void _hookDebugPrint(SuperLogConfig config) {
 // Hook debugPrint if enabled
 if (config.captureDebugPrint) {
  final originalDebugPrint = debugPrint;
  debugPrint = (String? message, {int? wrapWidth}) {
   if (!_isInDebugPrint && !_isCapturingPrint && isInitialized) {
    _isInDebugPrint = true;
    try {
     instance.addLog(message ?? '', level: LogLevel.debug);
    } catch (_) {
    } finally {
     _isInDebugPrint = false;
    }
   }
   originalDebugPrint(message, wrapWidth: wrapWidth);
  };
 }

 // Hook print() if enabled
 if (config.capturePrint) {
  runZoned(
   () {},
   zoneSpecification: ZoneSpecification(
    print: (Zone self, ZoneDelegate parent, Zone zone, String line) {
```

```
        if (!_isCapturingPrint && !_isInDebugPrint && isInitialized) {
          _isCapturingPrint = true;
          try {
            instance.addLog(line, level: LogLevel.info);
          } catch (_) {
          } finally {
            _isCapturingPrint = false;
          }
        }
        parent.print(zone, line);
      },
    ),
  );
 }
}

SuperLogManager._() {
 _initialize();
}

SuperLogManager._disabled() {
 _logs = <SuperLogEntry>[];
 _pendingLogs = <SuperLogEntry>[];
}

late final List<SuperLogEntry> _logs;
```

```
late final List<SuperLogEntry> _pendingLogs;
Timer? _batchTimer;

/// Get all logs (unmodifiable)
List<SuperLogEntry> get logs => List.unmodifiable(_logs);

void _initialize() {
  _logs = <SuperLogEntry>[];
  _pendingLogs = <SuperLogEntry>[];
}

void addLog(
  String message, {
  LogLevel level = LogLevel.info,
  String? tag,
  Object? error,
  StackTrace? stackTrace,
}) {
  final log = SuperLogEntry(
    message: message,
    level: level,
    tag: tag,
    timestamp: DateTime.now(),
    error: error,
    stackTrace: stackTrace,
  );
```

```
  _pendingLogs.add(log);

  if (_batchTimer?.isActive ?? false) {
    _batchTimer!.cancel();
  }

  _batchTimer = Timer(const Duration(milliseconds: 16), () {
    _processPendingLogs();
  });
}

void _processPendingLogs() {
  if (_pendingLogs.isEmpty) return;

  _logs.addAll(_pendingLogs);
  _pendingLogs.clear();

  if (_logs.length > (_config?.maxLogs ?? 1000)) {
    _logs.removeRange(0, _logs.length - (_config?.maxLogs ?? 1000));
  }

  _cachedFilteredLogs = null;
  _cachedSearchQuery = null;
  _cachedLevelFilter = null;
  _cachedLogsLength = null;
```

```
  notifyListeners();
}

List<SuperLogEntry>? _cachedFilteredLogs;
String? _cachedSearchQuery;
LogLevel? _cachedLevelFilter;
int? _cachedLogsLength;

List<SuperLogEntry> get filteredLogs {
  final logsLength = _logs.length;
  if (_cachedFilteredLogs != null &&
      _cachedLogsLength == logsLength &&
      _cachedSearchQuery == _searchQuery &&
      _cachedLevelFilter == _levelFilter) {
    return _cachedFilteredLogs!;
  }

  List<SuperLogEntry> filtered = _logs;

  if (_levelFilter != null) {
    filtered = filtered.where((log) => log.level == _levelFilter).toList();
  }

  if (_searchQuery.isNotEmpty) {
    final query = _searchQuery.toLowerCase();
```

```
   filtered = filtered
       .where(
        (log) =>
            log.message.toLowerCase().contains(query) ||
            (log.tag?.toLowerCase().contains(query) ?? false) ||
            log.level.name.toLowerCase().contains(query),
       )
       .toList();
  }

  _cachedFilteredLogs = filtered;
  _cachedLogsLength = logsLength;
  _cachedSearchQuery = _searchQuery;
  _cachedLevelFilter = _levelFilter;

  return filtered;
}

String _searchQuery = '';
LogLevel? _levelFilter;

String get searchQuery => _searchQuery;
LogLevel? get levelFilter => _levelFilter;

void setSearchQuery(String query) {
  if (_searchQuery != query) {
```

```
    _searchQuery = query;
    _cachedFilteredLogs = null;
    notifyListeners();
  }
}

void setLevelFilter(LogLevel? level) {
  if (_levelFilter != level) {
    _levelFilter = level;
    _cachedFilteredLogs = null;
    notifyListeners();
  }
}

void clearLogs() {
  _logs.clear();
  _pendingLogs.clear();
  _cachedFilteredLogs = null;
  _cachedSearchQuery = null;
  _cachedLevelFilter = null;
  _cachedLogsLength = null;
  notifyListeners();
}

int _cachedErrorCount = 0;
int? _cachedErrorCountLogsLength;
```

```
int get errorCount {
  final logsLength = _logs.length;
  if (_cachedErrorCountLogsLength == logsLength) {
    return _cachedErrorCount;
  }

  _cachedErrorCount = _logs
      .where((log) => log.level == LogLevel.error)
      .length;
  _cachedErrorCountLogsLength = logsLength;
  return _cachedErrorCount;
}

@override
void dispose() {
  _batchTimer?.cancel();
  _batchTimer = null;
  _pendingLogs.clear();
  _cachedFilteredLogs = null;
  super.dispose();
}

void deleteLog(SuperLogEntry log) {
  _logs.remove(log);
  _cachedFilteredLogs = null;
```

```
    _cachedErrorCountLogsLength = null;
    notifyListeners();
  }

  void deleteLogs(List<SuperLogEntry> logsToRemove) {
    _logs.removeWhere((log) => logsToRemove.contains(log));
    _cachedFilteredLogs = null;
    _cachedErrorCountLogsLength = null;
    notifyListeners();
  }

  static void reset() {
    _instance = null;
    _config = null;
  }
}
import 'package:flutter/material.dart';
import 'log_navigation_callbacks.dart';
import 'log_entry.dart';

/// Configuration for SuperLogManager.runApp
class SuperLogConfig {
  /// Enable or disable the debug tool completely
  final bool enabled;

  /// Maximum number of logs to keep in memory
```

```
final int maxLogs;

/// Show debug overlay bubble
final bool showOverlayBubble;

/// Auto-detect error level from message text
final bool autoDetectErrorLevel;

/// Capture debugPrint calls
final bool captureDebugPrint;

/// Capture print() calls via ZoneSpecification
final bool capturePrint;

/// Navigation callbacks for custom navigation systems
/// If null, uses default Flutter Navigator
final SuperLogNavigationCallbacks? navigationCallbacks;

/// Bubble size (diameter)
final double bubbleSize;

/// Initial bubble position (corner of the bubble)
/// Respects app text direction (RTL/LTR):
/// - RTL: dx = distance from right edge, dy = distance from top
/// - LTR: dx = distance from left edge, dy = distance from top
/// Offset(16, 100) means 16 pixels from the appropriate edge, 100 pixels from top
```

```
/// Default: Offset(16.0, 100.0)
final Offset initialBubblePosition;

/// Bubble color
final Color bubbleColor;

/// Bubble icon color
final Color bubbleIconColor;

/// Error badge color
final Color errorBadgeColor;

/// Error badge text color
final Color errorBadgeTextColor;

/// Enable drag to reposition bubble
final bool enableBubbleDrag;

/// Hide bubble when debug screen is open
final bool hideBubbleWhenScreenOpen;

/// Panel height as fraction of screen height (0.0 to 1.0)
/// Default: 0.6 (60% of screen height)
final double panelHeightFraction;

/// Dim background when overlay is open
```

```
final bool dimOverlayBackground;

/// Debug log screen route name (for named routes)
/// Default: '/super-debug-log'
final String debugLogRouteName;

/// Enable log filtering
final bool enableLogFiltering;

/// Enable log search
final bool enableLogSearch;

/// Enable log deletion
final bool enableLogDeletion;

/// Enable log export
final bool enableLogExport;

/// Default log level filter (null = show all)
final LogLevel? defaultLogLevelFilter;

/// Enable auto-scroll to latest log
final bool autoScrollToLatest;

/// Log screen theme mode (null = use system theme)
final ThemeMode? logScreenThemeMode;
```

```
/// Enable performance optimizations
final bool enablePerformanceOptimizations;

/// Log retention duration (null = keep all logs)
final Duration? logRetentionDuration;

/// Enable log compression for large logs
final bool enableLogCompression;

/// Maximum log message length (null = no limit)
final int? maxLogMessageLength;

/// Enable crash reporting integration
final bool enableCrashReporting;

/// Custom crash reporter callback
final void Function(Object error, StackTrace? stackTrace)? onCrashReport;

/// Enable network log capture
final bool enableNetworkLogCapture;

/// Enable database log capture
final bool enableDatabaseLogCapture;

/// Enable UI interaction log capture
```

```
final bool enableUIInteractionLogCapture;

/// Platform-specific settings
final Map<String, dynamic>? platformSettings;

const SuperLogConfig({
  this.enabled = true,
  this.maxLogs = 1000,
  this.showOverlayBubble = true,
  this.autoDetectErrorLevel = true,
  this.captureDebugPrint = true,
  this.capturePrint = true,
  this.navigationCallbacks,
  this.bubbleSize = 56.0,
  this.initialBubblePosition = const Offset(16.0, 100.0),
  this.bubbleColor = const Color(0xCCFF0000), // Red with opacity
  this.bubbleIconColor = Colors.white,
  this.errorBadgeColor = Colors.red,
  this.errorBadgeTextColor = Colors.white,
  this.enableBubbleDrag = true,
  this.hideBubbleWhenScreenOpen = true,
  this.panelHeightFraction = 0.6,
  this.dimOverlayBackground = true,
  this.debugLogRouteName = '/super-debug-log',
  this.enableLogFiltering = true,
  this.enableLogSearch = true,
```

```
  this.enableLogDeletion = true,
  this.enableLogExport = true,
  this.defaultLogLevelFilter,
  this.autoScrollToLatest = true,
  this.logScreenThemeMode,
  this.enablePerformanceOptimizations = true,
  this.logRetentionDuration,
  this.enableLogCompression = false,
  this.maxLogMessageLength,
  this.enableCrashReporting = false,
  this.onCrashReport,
  this.enableNetworkLogCapture = false,
  this.enableDatabaseLogCapture = false,
  this.enableUIInteractionLogCapture = false,
  this.platformSettings,
});

/// Disabled configuration (tool completely ignored)
const SuperLogConfig.disabled()
  : enabled = false,
    maxLogs = 0,
    showOverlayBubble = false,
    autoDetectErrorLevel = false,
    captureDebugPrint = false,
    capturePrint = false,
    navigationCallbacks = null,
```

```
bubbleSize = 56.0,
initialBubblePosition = const Offset(16.0, 100.0),
bubbleColor = const Color(0xCCFF0000),
bubbleIconColor = Colors.white,
errorBadgeColor = Colors.red,
errorBadgeTextColor = Colors.white,
enableBubbleDrag = true,
hideBubbleWhenScreenOpen = true,
panelHeightFraction = 0.6,
dimOverlayBackground = true,
debugLogRouteName = '/super-debug-log',
enableLogFiltering = true,
enableLogSearch = true,
enableLogDeletion = true,
enableLogExport = true,
defaultLogLevelFilter = null,
autoScrollToLatest = true,
logScreenThemeMode = null,
enablePerformanceOptimizations = true,
logRetentionDuration = null,
enableLogCompression = false,
maxLogMessageLength = null,
enableCrashReporting = false,
onCrashReport = null,
enableNetworkLogCapture = false,
enableDatabaseLogCapture = false,
```

```
    enableUIInteractionLogCapture = false,
    platformSettings = null;
}
enum LogLevel { info, warning, error, debug }

/// Optimized log entry with cached lowercase strings for filtering
class SuperLogEntry {
 final String message;
 final DateTime timestamp;
 final LogLevel level;
 final String? tag;
 final Object? error;
 final StackTrace? stackTrace;

 // Cache lowercase strings for efficient filtering
 late final String _lowerMessage;
 late final String? _lowerTag;
 late final String _lowerLevelName;

 SuperLogEntry({
  required this.message,
  required this.timestamp,
  this.level = LogLevel.info,
  this.tag,
  this.error,
  this.stackTrace,
```

```
  }) {
    // Pre-compute lowercase strings once
    _lowerMessage = message.toLowerCase();
    _lowerTag = tag?.toLowerCase();
    _lowerLevelName = level.name.toLowerCase();
  }

  /// Efficient filter matching using pre-computed lowercase strings
  bool matchesFilter(String query) {
    if (query.isEmpty) return true;
    final lowerQuery = query.toLowerCase();
    return _lowerMessage.contains(lowerQuery) ||
        (_lowerTag?.contains(lowerQuery) ?? false) ||
        _lowerLevelName.contains(lowerQuery);
  }
}import 'package:flutter/material.dart';

import '../controllers/super_log_manager.dart';

/// Navigation callbacks for custom navigation systems
///
/// - For MaterialApp: Not needed (uses MaterialApp.builder automatically)
/// - For GetX: Not needed (GetX handles navigation automatically)
/// - For go_router: Provide callbacks using router.push/pop
class SuperLogNavigationCallbacks {
  /// Navigate to debug log screen
```

```
/// [debugLogScreenBuilder] provides the DebugLogScreen widget
/// Return a Future that completes when navigation is done
///
/// Example for go_router:
/// ```dart
/// SuperLogNavigationCallbacks(
///   onNavigateToDebugLog: (context, builder) => router.push('/debug-log'),
///   onNavigateBack: (context) => router.pop(),
/// )
/// ```
final Future<void> Function(
  BuildContext? context,
  Widget Function() debugLogScreenBuilder,
)?
onNavigateToDebugLog;

/// Navigate back
/// Example for go_router: `(context) => router.pop()`
final void Function(BuildContext? context)? onNavigateBack;

/// Show snackbar/toast message
/// Example: `(context, msg) => ScaffoldMessenger.of(context).showSnackBar(...)`
final void Function(BuildContext? context, String message)? onShowSnackbar;

/// Check if can navigate back
/// Example for go_router: `(context) => router.canPop()`
```

```
final bool Function(BuildContext? context)? canNavigateBack;


const SuperLogNavigationCallbacks({
 this.onNavigateToDebugLog,
 this.onNavigateBack,
 this.onShowSnackbar,
 this.canNavigateBack,
});


/// Default Flutter Navigator callbacks
/// Used when MaterialApp.builder is not available
/// For MaterialApp, use builder injection (automatic)
factory SuperLogNavigationCallbacks.flutter() {
 return SuperLogNavigationCallbacks(
  onNavigateToDebugLog: (context, debugLogScreenBuilder) async {
   final navigator = _resolveNavigator(context);
   if (navigator == null) return;
   await navigator.push(
    MaterialPageRoute(
     builder: (_) => debugLogScreenBuilder(),
     fullscreenDialog: true,
    ),
   );
  },
  onNavigateBack: (context) {
   final navigator = _resolveNavigator(context);
```

```
      if (navigator == null) return;
      if (navigator.canPop()) {
        navigator.pop();
      }
    },
    onShowSnackbar: (context, message) {
      final messengerContext = context ?? SuperLogManager.navigatorKey.currentContext;
      if (messengerContext == null) return;
      ScaffoldMessenger.of(messengerContext).showSnackBar(
        SnackBar(
          content: Text(message),
          duration: const Duration(seconds: 2),
        ),
      );
    },
    canNavigateBack: (context) {
      final navigator = _resolveNavigator(context);
      if (navigator == null) return false;
      return navigator.canPop();
    },
  );
}

/// Legacy method for backward compatibility
factory SuperLogNavigationCallbacks.flutterDefaultNavigator() {
  return SuperLogNavigationCallbacks.flutter();
```

```
}

/// GetX navigation callbacks
/// Note: GetX usually doesn't need callbacks as it handles navigation automatically
/// This is provided for custom GetX setups
factory SuperLogNavigationCallbacks.getX() {
  // GetX doesn't need callbacks - it handles navigation automatically
  // Return empty callbacks
  return const SuperLogNavigationCallbacks();
}

/// go_router navigation callbacks
///
/// Users should provide callbacks manually:
/// ```dart
/// final router = GoRouter(...);
/// SuperLogNavigationCallbacks(
///   onNavigateToDebugLog: (context, builder) => router.push('/debug-log'),
///   onNavigateBack: (context) => router.pop(),
///   canNavigateBack: (context) => router.canPop(),
/// )
/// ```
///
/// This factory is kept for backward compatibility but users should
/// provide callbacks directly for better type safety.
@Deprecated('Provide callbacks directly for go_router')
```

```
factory SuperLogNavigationCallbacks.goRouter(dynamic router) {
  return SuperLogNavigationCallbacks(
    onNavigateToDebugLog: (context, debugLogScreenBuilder) async {
      if (context == null) return;
      // Try to use router if it has push method
      try {
        if (router != null) {
          // Assume router has push method (go_router pattern)
          await (router as dynamic).push('/debug-log');
          return;
        }
      } catch (e) {
        // Fall through to Navigator fallback
      }
      // Fallback to Navigator
      final navigator = _resolveNavigator(context);
      if (navigator == null) return;
      await navigator.push(
        MaterialPageRoute(
          builder: (_) => debugLogScreenBuilder(),
          fullscreenDialog: true,
        ),
      );
    },
    onNavigateBack: (context) {
      if (context == null) return;
```

```
  try {
    if (router != null) {
      (router as dynamic).pop();
      return;
    }
  } catch (e) {
    // Fall through to Navigator fallback
  }
  final navigator = _resolveNavigator(context);
  if (navigator == null) return;
  if (navigator.canPop()) {
    navigator.pop();
  }
},
onShowSnackbar: (context, message) {
  if (context == null) return;
  ScaffoldMessenger.of(context).showSnackBar(
    SnackBar(
      content: Text(message),
      duration: const Duration(seconds: 2),
    ),
  );
},
canNavigateBack: (context) {
  if (context == null) return false;
  try {
```

```
        if (router != null) {
          return (router as dynamic).canPop() ?? false;
        }
      } catch (e) {
        // Fall through to Navigator fallback
      }
      final navigator = _resolveNavigator(context);
      if (navigator == null) return false;
      return navigator.canPop();
    },
  );
 }
}

NavigatorState? _resolveNavigator(BuildContext? context) {
 final navigatorFromKey = SuperLogManager.navigatorKey.currentState;
 if (navigatorFromKey != null) return navigatorFromKey;
 if (context == null) return null;
 return Navigator.maybeOf(context, rootNavigator: true);
}
import 'package:flutter/material.dart';
import 'package:flutter/services.dart';
import '../controllers/super_log_manager.dart';
import '../models/log_entry.dart';

/// Debug log screen using pure Flutter state management
```

```
/// Uses LogManager.instance directly (singleton pattern)
///
/// No navigation callbacks needed - uses onClose callback for overlay-based navigation
class SuperDebugLogScreen extends StatefulWidget {
  const SuperDebugLogScreen({super.key, this.onClose});

  /// Optional callback to close the screen (used when not in navigator context)
  final VoidCallback? onClose;

  @override
  State<SuperDebugLogScreen> createState() => _SuperDebugLogScreenState();
}

class _SuperDebugLogScreenState extends State<SuperDebugLogScreen> {
  late final SuperLogManager _logManager;
  final ValueNotifier<double> _fontSize = ValueNotifier<double>(16.0);
  final ValueNotifier<Set<SuperLogEntry>> _selectedLogs =
      ValueNotifier<Set<SuperLogEntry>>(<SuperLogEntry>{});
  final ValueNotifier<bool> _isSelectionMode = ValueNotifier<bool>(false);

  @override
  void initState() {
    super.initState();
    _logManager = SuperLogManager.instance;
  }
```

```
// Cache dropdown items to avoid rebuilding
static const List<DropdownMenuItem<double>> _fontSizeItems = [
  DropdownMenuItem(value: 15.0, child: Text('15')),
  DropdownMenuItem(value: 16.0, child: Text('16')),
  DropdownMenuItem(value: 17.0, child: Text('17')),
  DropdownMenuItem(value: 18.0, child: Text('18')),
  DropdownMenuItem(value: 19.0, child: Text('19')),
  DropdownMenuItem(value: 20.0, child: Text('20')),
];

@override
void dispose() {
  _fontSize.dispose();
  _selectedLogs.dispose();
  _isSelectionMode.dispose();
  super.dispose();
}

void _toggleSelection(SuperLogEntry log) {
  final current = Set<SuperLogEntry>.from(_selectedLogs.value);
  if (current.contains(log)) {
    current.remove(log);
    if (current.isEmpty) {
      _isSelectionMode.value = false;
    }
  } else {
```

```
    current.add(log);
    _isSelectionMode.value = true;
  }
  _selectedLogs.value = current;
}

void _selectAll() {
  final allLogs = _logManager.filteredLogs;
  _selectedLogs.value = Set<SuperLogEntry>.from(allLogs);
  _isSelectionMode.value = true;
}

void _clearSelection() {
  _selectedLogs.value = <SuperLogEntry>{};
  _isSelectionMode.value = false;
}

Future<void> _deleteSelected() async {
  if (_selectedLogs.value.isEmpty) return;

  final confirmed = await _showDeleteConfirmationDialog(
    context,
    'Delete ${_selectedLogs.value.length} selected log(s)?',
    'This action cannot be undone.',
  );
```

```
  if (confirmed == true) {
    _logManager.deleteLogs(_selectedLogs.value.toList());
    _clearSelection();
  }
}

Future<bool?> _showDeleteConfirmationDialog(
  BuildContext context,
  String title,
  String content,
) async {
  return showDialog<bool>(
    context: context,
    builder: (context) => AlertDialog(
      title: Text(title),
      content: Text(content),
      actions: [
        TextButton(
          onPressed: () => Navigator.of(context).pop(false),
          child: const Text('Cancel'),
        ),
        TextButton(
          onPressed: () => Navigator.of(context).pop(true),
          style: TextButton.styleFrom(foregroundColor: Colors.red),
          child: const Text('Delete'),
        ),
```

```
      ],
    ),
  );
}

Future<void> _clearAllLogs(BuildContext context) async {
  final confirmed = await _showDeleteConfirmationDialog(
    context,
    'Clear all logs?',
    'This will delete all logs. This action cannot be undone.',
  );

  if (confirmed == true) {
    _logManager.clearLogs();
  }
}

Future<void> _deleteSingleLog(BuildContext context, SuperLogEntry log) async {
  final confirmed = await _showDeleteConfirmationDialog(
    context,
    'Delete this log?',
    'This action cannot be undone.',
  );

  if (confirmed == true) {
    _logManager.deleteLog(log);
```

```
  }
}

void _copySelected(BuildContext context) {
  if (_selectedLogs.value.isEmpty) return;

  final text = _selectedLogs.value
      .map((l) => '${l.timestamp} ${l.message}')
      .join('\n\n');

  Clipboard.setData(ClipboardData(text: text));

  _showSnackbar(
    context,
    '${_selectedLogs.value.length} logs copied to clipboard',
  );

  _clearSelection();
}

void _showSnackbar(BuildContext context, String message) {
  ScaffoldMessenger.of(context).showSnackBar(
    SnackBar(
      content: Text(message),
      duration: const Duration(seconds: 2),
      behavior: SnackBarBehavior.floating,
```

```
    ),
  );
}

void _handleBack() {
  if (widget.onClose != null) {
    // Close via callback (overlay mode)
    widget.onClose!();
  } else {
    // Try Navigator.pop (navigator mode)
    if (Navigator.canPop(context)) {
      Navigator.of(context).pop();
    }
  }
}

@override
Widget build(BuildContext context) {
  final theme = Theme.of(context);
  final isDark = theme.brightness == Brightness.dark;

  return Scaffold(
    appBar: _buildAppBar(context, theme, isDark),
    body: Column(
      children: [
        _buildFilterBar(context, theme, isDark),
```

```
Expanded(
  child: AnimatedBuilder(
    animation: _logManager,
    builder: (context, child) {
      final logs = _logManager.filteredLogs;
      final logsLength = logs.length;

      if (logsLength == 0) {
        return Center(
          child: Column(
            mainAxisAlignment: MainAxisAlignment.center,
            children: [
              Icon(
                Icons.info_outline,
                size: 64,
                color: theme.hintColor,
              ),
              const SizedBox(height: 16),
              Text(
                'No logs found',
                style: theme.textTheme.titleMedium?.copyWith(
                  color: theme.hintColor,
                ),
              ),
            ],
          ),
```

```
  );
}

return ListView.separated(
  key: const PageStorageKey('debug_logs_list'),
  itemCount: logsLength,
  cacheExtent: 500,
  addAutomaticKeepAlives: false,
  addRepaintBoundaries: true,
  addSemanticIndexes: false,
  separatorBuilder: (_, __) => Divider(
    height: 1,
    thickness: 0.5,
    color: isDark ? Colors.grey.shade800 : Colors.grey.shade300,
  ),
  itemBuilder: (context, index) {
    final log = logs[index];
    return _LogItem(
      key: ValueKey(
        '${log.timestamp.millisecondsSinceEpoch}_${log.message.hashCode}',
      ),
      log: log,
      fontSize: _fontSize,
      selectedLogs: _selectedLogs,
      isSelectionMode: _isSelectionMode,
      isDark: isDark,
```

```
                theme: theme,
                onTap: () {
                  if (_isSelectionMode.value) {
                    _toggleSelection(log);
                  }
                },
                onLongPress: () => _toggleSelection(log),
                onCopy: () {
                  final text = '${log.timestamp} ${log.message}';
                  Clipboard.setData(ClipboardData(text: text));
                  _showSnackbar(context, 'Log entry copied');
                },
                onDelete: () => _deleteSingleLog(context, log),
              );
            },
          );
        },
      ),
    ),
  ],
 ),
 );
}

PreferredSizeWidget _buildAppBar(
  BuildContext context,
```

```
  ThemeData theme,
  bool isDark,
) {
  return PreferredSize(
    preferredSize: const Size.fromHeight(kToolbarHeight),
    child: ValueListenableBuilder<bool>(
      valueListenable: _isSelectionMode,
      builder: (context, isSelectionMode, child) {
        return ValueListenableBuilder<Set<SuperLogEntry>>(
          valueListenable: _selectedLogs,
          builder: (context, selectedLogs, child) {
            return AppBar(
              backgroundColor: isDark
                  ? theme.appBarTheme.backgroundColor ??
                        theme.colorScheme.surface
                  : theme.appBarTheme.backgroundColor ??
                        theme.colorScheme.surface,
              foregroundColor:
                  theme.appBarTheme.foregroundColor ??
                  theme.colorScheme.onSurface,
              title: isSelectionMode
                  ? Text('${selectedLogs.length} selected')
                  : const Text('Debug Logs'),
              leading: IconButton(
                icon: Icon(isSelectionMode ? Icons.close : Icons.arrow_back),
                onPressed: isSelectionMode ? _clearSelection : _handleBack,
```

```
            tooltip: isSelectionMode ? 'Cancel Selection' : 'Back',
          ),
          actions: _buildAppBarActions(context, theme, isDark),
        );
      },
    );
  },
 ),
);
}

List<Widget> _buildAppBarActions(
 BuildContext context,
 ThemeData theme,
 bool isDark,
) {
 final iconColor =
   theme.appBarTheme.iconTheme?.color ?? theme.colorScheme.onSurface;
 final textColor =
   theme.appBarTheme.titleTextStyle?.color ?? theme.colorScheme.onSurface;

 return [
  ValueListenableBuilder<bool>(
   valueListenable: _isSelectionMode,
   builder: (context, isSelectionMode, child) {
    if (isSelectionMode) {
```

```
return ValueListenableBuilder<Set<SuperLogEntry>>(
 valueListenable: _selectedLogs,
 builder: (context, selectedLogs, child) {
  final filteredCount = _logManager.filteredLogs.length;
  final selectedCount = selectedLogs.length;
  return Row(
   mainAxisSize: MainAxisSize.min,
   children: [
    if (selectedCount < filteredCount)
     IconButton(
      icon: const Icon(Icons.select_all),
      onPressed: _selectAll,
      tooltip: 'Select All',
     ),
    if (selectedCount > 0) ...[
     IconButton(
      icon: const Icon(Icons.copy),
      onPressed: () => _copySelected(context),
      tooltip: 'Copy Selected',
     ),
     IconButton(
      icon: const Icon(Icons.delete),
      onPressed: _deleteSelected,
      tooltip: 'Delete Selected',
     ),
    ],
```

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

// Default Mode Actions
return Row(
 mainAxisSize: MainAxisSize.min,
 children: [
  // Font Size Selector
  ValueListenableBuilder<double>(
   valueListenable: _fontSize,
   builder: (context, fontSize, child) {
    return DropdownButton<double>(
     dropdownColor: theme.colorScheme.surface,
     value: fontSize,
     style: TextStyle(color: textColor),
     icon: Icon(Icons.text_fields, color: iconColor),
     underline: Container(),
     items: _fontSizeItems
       .map(
        (item) => DropdownMenuItem<double>(
         value: item.value,
         child: Text(
          item.value.toString(),
```

```
                    style: TextStyle(color: textColor),
                  ),
                ),
              )
              .toList(),
          onChanged: (val) {
            if (val != null) _fontSize.value = val;
          },
        );
      },
    ),
    IconButton(
      icon: const Icon(Icons.delete_sweep),
      onPressed: () => _clearAllLogs(context),
      tooltip: 'Clear All',
    ),
  ],
 );
 },
 ),
];
}

Widget _buildFilterBar(BuildContext context, ThemeData theme, bool isDark) {
 final surfaceColor = theme.colorScheme.surface;
 final onSurfaceColor = theme.colorScheme.onSurface;
```

```
final hintColor = theme.hintColor;

return Container(
 color: theme.colorScheme.surfaceContainerHighest,
 padding: const EdgeInsets.all(8.0),
 child: Row(
  children: [
   Expanded(
    child: TextField(
     style: TextStyle(color: onSurfaceColor),
     decoration: InputDecoration(
      hintText: 'Search logs...',
      hintStyle: TextStyle(color: hintColor),
      prefixIcon: Icon(Icons.search, color: hintColor),
      border: OutlineInputBorder(
       borderRadius: BorderRadius.circular(8),
       borderSide: BorderSide.none,
      ),
      filled: true,
      fillColor: surfaceColor,
      contentPadding: const EdgeInsets.symmetric(
       horizontal: 10,
       vertical: 0,
      ),
     ),
     onChanged: (val) => _logManager.setSearchQuery(val),
```

```
 ),
),
const SizedBox(width: 8),
Container(
 padding: const EdgeInsets.symmetric(horizontal: 12),
 decoration: BoxDecoration(
  color: surfaceColor,
  borderRadius: BorderRadius.circular(8),
 ),
 child: DropdownButtonHideUnderline(
  child: DropdownButton<LogLevel?>(
   value: _logManager.levelFilter,
   hint: Text('Level', style: TextStyle(color: hintColor)),
   icon: Icon(Icons.filter_list, color: onSurfaceColor),
   dropdownColor: theme.colorScheme.surface,
   style: TextStyle(color: onSurfaceColor),
   items: [
    DropdownMenuItem(
     value: null,
     child: Text('All', style: TextStyle(color: onSurfaceColor)),
    ),
    ...LogLevel.values.map(
     (l) => DropdownMenuItem(
      value: l,
      child: _buildLevelBadge(l, theme, isDark),
     ),
```

```
            ),
          ],
          onChanged: (val) => _logManager.setLevelFilter(val),
        ),
      ),
    ),
  ],
 ),
);
}

Widget _buildLevelBadge(LogLevel level, ThemeData theme, bool isDark) {
 final color = _getLevelColor(level, isDark);
 return Text(
  level.name.toUpperCase(),
  style: TextStyle(color: color, fontWeight: FontWeight.bold),
 );
}

Color _getLevelColor(LogLevel level, bool isDark) {
 switch (level) {
  case LogLevel.error:
   return isDark ? Colors.red.shade400 : Colors.red.shade700;
  case LogLevel.warning:
   return isDark ? Colors.orange.shade400 : Colors.orange.shade700;
  case LogLevel.debug:
```

```
      return isDark ? Colors.blue.shade400 : Colors.blue.shade700;
    case LogLevel.info:
      return isDark ? Colors.green.shade400 : Colors.green.shade700;
   }
  }
}

// ... (Rest of _LogItem class remains exactly the same)
class _LogItem extends StatefulWidget {
  final SuperLogEntry log;
  final ValueNotifier<double> fontSize;
  final ValueNotifier<Set<SuperLogEntry>> selectedLogs;
  final ValueNotifier<bool> isSelectionMode;
  final bool isDark;
  final ThemeData theme;
  final VoidCallback onTap;
  final VoidCallback onLongPress;
  final VoidCallback onCopy;
  final VoidCallback onDelete;

  const _LogItem({
   super.key,
   required this.log,
   required this.fontSize,
   required this.selectedLogs,
   required this.isSelectionMode,
```

```
    required this.isDark,
    required this.theme,
    required this.onTap,
    required this.onLongPress,
    required this.onCopy,
    required this.onDelete,
  });

  @override
  State<_LogItem> createState() => _LogItemState();
}

class _LogItemState extends State<_LogItem> {
  bool _expanded = false;
  late final Color _textColor;
  late final Color _tagColor;

  @override
  void initState() {
    super.initState();
    _textColor = _getTextColorForLevel(widget.log.level, widget.isDark);
    _tagColor = _getTagColorForLevel(widget.log.level, widget.isDark);
  }

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

```
return ValueListenableBuilder<bool>(
 valueListenable: widget.isSelectionMode,
 builder: (context, isSelectionMode, child) {
  return ValueListenableBuilder<Set<SuperLogEntry>>(
   valueListenable: widget.selectedLogs,
   builder: (context, selectedLogs, child) {
    return ValueListenableBuilder<double>(
     valueListenable: widget.fontSize,
     builder: (context, currentFontSize, child) {
      final isSelected = selectedLogs.contains(widget.log);
      final bgColor = isSelected
        ? (widget.isDark
            ? Colors.blue.shade900.withOpacity(0.3)
            : Colors.blue.shade100)
        : _getBackgroundColorForLevel(
            widget.log.level,
            widget.isDark,
            widget.theme,
          );

      return Directionality(
       textDirection: TextDirection.ltr,
       child: Container(
        color: bgColor,
        padding: const EdgeInsets.symmetric(
         horizontal: 12,
```

```
  vertical: 8,
 ),
 child: InkWell(
  onTap: widget.onTap,
  onLongPress: widget.onLongPress,
  child: Row(
   crossAxisAlignment: CrossAxisAlignment.start,
   children: [
    if (isSelectionMode)
     Padding(
      padding: const EdgeInsets.only(right: 12, top: 4),
      child: Icon(
       isSelected
         ? Icons.check_circle
         : Icons.radio_button_unchecked,
       color: isSelected
         ? (widget.isDark
           ? Colors.blue.shade300
           : Colors.blue.shade700)
         : (widget.isDark
           ? Colors.grey.shade600
           : Colors.grey.shade400),
       size: 20,
      ),
     ),
    Expanded(
```

```
child: Column(
 crossAxisAlignment: CrossAxisAlignment.start,
 children: [
  Row(
   children: [
    Text(
     _formatTimestamp(widget.log.timestamp),
     style: TextStyle(
      fontSize: 12,
      color: widget.isDark
        ? Colors.grey.shade400
        : Colors.grey.shade600,
     ),
    ),
    const Spacer(),
    if (widget.log.tag != null)
     Container(
      padding: const EdgeInsets.symmetric(
       horizontal: 6,
       vertical: 2,
      ),
      decoration: BoxDecoration(
       color: _tagColor,
       borderRadius: BorderRadius.circular(
        4,
       ),
```

```
    ),
    child: Text(
      widget.log.tag!,
      style: TextStyle(
        fontSize: 11,
        fontWeight: FontWeight.bold,
        color: _textColor,
      ),
    ),
  ),
if (!isSelectionMode) ...[
  const SizedBox(width: 8),
  GestureDetector(
    onTap: widget.onCopy,
    child: Icon(
      Icons.copy,
      size: 16,
      color: widget.isDark
          ? Colors.grey.shade400
          : Colors.grey.shade600,
    ),
  ),
  const SizedBox(width: 8),
  GestureDetector(
    onTap: widget.onDelete,
    child: Icon(
```

```
        Icons.delete_outline,
        size: 16,
        color: widget.isDark
            ? Colors.grey.shade400
            : Colors.grey.shade600,
      ),
    ),
  ],
 ],
),
const SizedBox(height: 4),
_buildMessage(_textColor, currentFontSize),
if (widget.log.error != null) ...[
 const SizedBox(height: 4),
 Container(
  padding: const EdgeInsets.all(8),
  decoration: BoxDecoration(
   color: widget.isDark
       ? Colors.red.shade900.withOpacity(0.3)
       : Colors.red.shade100,
   borderRadius: BorderRadius.circular(4),
  ),
  child: Text(
   widget.log.error.toString(),
   style: TextStyle(
    color: widget.isDark
```

```
                    ? Colors.red.shade300
                    : Colors.red.shade900,
                fontWeight: FontWeight.bold,
                fontSize: 12,
              ),
            ),
          ),
        ],
      ),
    ),
  ),
 ],
),
),
),
),
);
},
);
},
);
},
);
}

Widget _buildMessage(Color textColor, double fontSize) {
  const int maxLinesCollapsed = 3;
```

```
final textStyle = TextStyle(
  color: textColor,
  fontFamily: 'Courier',
  fontSize: fontSize,
  height: 1.2,
);

final message = widget.log.message;
final isShort = message.length < 150 && message.split('\n').length < 3;

if (isShort) {
  return Text(message, style: textStyle);
}

return Column(
  crossAxisAlignment: CrossAxisAlignment.start,
  children: [
    Text(
      message,
      style: textStyle,
      maxLines: _expanded ? null : maxLinesCollapsed,
      overflow: _expanded ? TextOverflow.visible : TextOverflow.ellipsis,
    ),
    GestureDetector(
      onTap: () => setState(() => _expanded = !_expanded),
      child: Padding(
```

```
        padding: const EdgeInsets.only(top: 4.0),
        child: Text(
         _expanded ? 'Show less' : 'Show more',
         style: TextStyle(
           color: widget.isDark
              ? Colors.blue.shade300
              : Colors.blue.shade700,
           fontSize: 12,
           fontWeight: FontWeight.bold,
         ),
        ),
       ),
     ),
   ],
  );
}

String _formatTimestamp(DateTime timestamp) {
  final h = timestamp.hour.toString().padLeft(2, '0');
  final m = timestamp.minute.toString().padLeft(2, '0');
  final s = timestamp.second.toString().padLeft(2, '0');
  final ms = timestamp.millisecond.toString().padLeft(3, '0');
  return '$h:$m:$s.$ms';
}

Color _getBackgroundColorForLevel(
```

```
  LogLevel level,
  bool isDark,
  ThemeData theme,
) {
  if (isDark) {
    switch (level) {
      case LogLevel.error:
        return Colors.red.shade900.withOpacity(0.2);
      case LogLevel.warning:
        return Colors.orange.shade900.withOpacity(0.2);
      case LogLevel.debug:
        return Colors.blue.shade900.withOpacity(0.2);
      case LogLevel.info:
        return theme.colorScheme.surface;
    }
  } else {
    switch (level) {
      case LogLevel.error:
        return Colors.red.shade50;
      case LogLevel.warning:
        return Colors.orange.shade50;
      case LogLevel.debug:
        return Colors.blue.shade50;
      case LogLevel.info:
        return Colors.white;
    }
```

```
  }
}

Color _getTextColorForLevel(LogLevel level, bool isDark) {
  if (isDark) {
    switch (level) {
      case LogLevel.error:
        return Colors.red.shade300;
      case LogLevel.warning:
        return Colors.orange.shade300;
      case LogLevel.debug:
        return Colors.blue.shade300;
      case LogLevel.info:
        return Colors.grey.shade300;
    }
  } else {
    switch (level) {
      case LogLevel.error:
        return Colors.red.shade900;
      case LogLevel.warning:
        return Colors.orange.shade900;
      case LogLevel.debug:
        return Colors.blue.shade900;
      case LogLevel.info:
        return Colors.grey.shade800;
    }
```

```
  }
}

Color _getTagColorForLevel(LogLevel level, bool isDark) {
  if (isDark) {
    switch (level) {
      case LogLevel.error:
        return Colors.red.shade800.withOpacity(0.4);
      case LogLevel.warning:
        return Colors.orange.shade800.withOpacity(0.4);
      case LogLevel.debug:
        return Colors.blue.shade800.withOpacity(0.4);
      case LogLevel.info:
        return Colors.grey.shade700;
    }
  } else {
    switch (level) {
      case LogLevel.error:
        return Colors.red.shade200;
      case LogLevel.warning:
        return Colors.orange.shade200;
      case LogLevel.debug:
        return Colors.blue.shade200;
      case LogLevel.info:
        return Colors.grey.shade300;
    }
```

```
  }
 }
}
import 'package:flutter/material.dart';
import 'package:flutter/services.dart';
import '../controllers/super_log_manager.dart';
import '../models/log_config.dart';
import 'debug_log_screen.dart';

/// Draggable debug overlay bubble widget
/// Uses pure Flutter with optimized performance techniques:
/// - GestureDetector for drag handling
/// - ValueNotifier for position updates (minimal rebuilds)
/// - Constrained position within screen bounds
/// - Navigator for debug screen navigation
///
/// [logManager] is optional - if not provided, uses LogManager.instance (singleton)
/// [navigatorContext] is optional - if provided, uses this context for navigation (from MaterialApp.builder)
class SuperDebugOverlayBubble extends StatefulWidget {
 final SuperLogManager? logManager;
 final VoidCallback? onTap;
 final bool hideWhenScreenOpen;
 final BuildContext? navigatorContext;

 const SuperDebugOverlayBubble({
```

```
    super.key,
    this.logManager,
    this.onTap,
    this.hideWhenScreenOpen = true,
    this.navigatorContext,
  });

  @override
  State<SuperDebugOverlayBubble> createState() =>
      _SuperDebugOverlayBubbleState();
}

class _SuperDebugOverlayBubbleState extends State<SuperDebugOverlayBubble> {
  late ValueNotifier<Offset> _position;
  final ValueNotifier<int> _errorCount = ValueNotifier<int>(0);
  final ValueNotifier<bool> _isOverlayOpen = ValueNotifier<bool>(false);

  Size _screenSize = Size.zero;
  bool _isDragging = false;
  Offset _dragStartPosition = Offset.zero;
  TextDirection _textDirection = TextDirection.ltr;

  late final SuperLogManager _logManager;
  late final SuperLogConfig _config;

  double get _bubbleSize => _config.bubbleSize;
```

```
@override
void initState() {
 super.initState();
 // Use provided logManager or fallback to singleton instance
 _logManager = widget.logManager ?? SuperLogManager.instance;
 // Get config
 _config = SuperLogManager.config ?? const SuperLogConfig();
 // Listen to log changes to update error count
 _logManager.addListener(_updateErrorCount);
 _updateErrorCount();
 // Initialize position - will be set based on config in first build
 _position = ValueNotifier<Offset>(Offset.zero);
}

@override
void dispose() {
 _isOverlayOpen.value = false;
 _logManager.removeListener(_updateErrorCount);
 _position.dispose();
 _errorCount.dispose();
 _isOverlayOpen.dispose();
 super.dispose();
}

void _updateErrorCount() {
```

```
  // Use optimized errorCount getter instead of filtering all logs
  final count = _logManager.errorCount;
  if (_errorCount.value != count) {
    _errorCount.value = count;
  }
}

void _onPanStart(DragStartDetails details) {
  _isDragging = false;
  _dragStartPosition = details.globalPosition;
}

void _onPanUpdate(DragUpdateDetails details, Size screenSize) {
  // Only start dragging if movement exceeds a threshold
  if (!_isDragging &&
      (_dragStartPosition - details.globalPosition).distance > 5) {
    _isDragging = true;
  }

  if (_isDragging) {
    final newPosition = _position.value + details.delta;

    // Constrain position within screen bounds (using bubble radius)
    final bubbleRadius = _bubbleSize / 2;
    final constrainedX = newPosition.dx.clamp(
      bubbleRadius,
```

```
      screenSize.width - bubbleRadius,
    );
    final constrainedY = newPosition.dy.clamp(
      bubbleRadius,
      screenSize.height - bubbleRadius,
    );

    _position.value = Offset(constrainedX, constrainedY);
  }
}

void _onPanEnd(DragEndDetails details) {
  if (_isDragging) {
    // Snap to nearest horizontal edge
    _snapToEdge();
  }
  _isDragging = false;
}

void _snapToEdge() {
  final screenWidth = _screenSize.width;
  final bubbleRadius = _bubbleSize / 2;
  final currentX = _position.value.dx;

  // Determine which edge is closer
  final distanceToLeft = currentX - bubbleRadius;
```

```
  final distanceToRight = screenWidth - currentX - bubbleRadius;

  // Snap to edge based on text direction preference
  // RTL: prefer right edge, LTR: prefer left edge
  final newX = _textDirection == TextDirection.rtl
      ? (distanceToRight < distanceToLeft
          ? screenWidth -
              bubbleRadius -
              16 // Snap to right with padding
          : bubbleRadius + 16) // Snap to left with padding
      : (distanceToLeft < distanceToRight
          ? bubbleRadius +
              16 // Snap to left with padding
          : screenWidth - bubbleRadius - 16); // Snap to right with padding

  _position.value = Offset(newX, _position.value.dy);
}

void _onTap() {
  debugPrint('SuperLogManager: Bubble tapped, isDragging: $_isDragging');
  if (_isDragging) {
    debugPrint('SuperLogManager: Ignoring tap - was dragging');
    return; // Ignore tap if we just finished dragging
  }

  if (widget.onTap != null) {
```

```
    debugPrint('SuperLogManager: Using custom onTap callback');
    widget.onTap!();
  } else {
    if (_isOverlayOpen.value) {
      debugPrint('SuperLogManager: Overlay is open, closing...');
      _closeOverlay();
    } else {
      debugPrint('SuperLogManager: Opening overlay...');
      _showOverlay();
    }
  }
}

void _onLongPress() {
  if (!mounted) return;

  // Copy all log messages to clipboard
  final allLogs = _logManager.logs;
  final logText = allLogs
      .map((log) {
        final timestamp = log.timestamp.toString().substring(0, 19);
        final level = log.level.name.toUpperCase();
        final tag = log.tag != null ? '[${log.tag}]' : '';
        return '[$timestamp] $level $tag ${log.message}';
      })
      .join('\n');
```

```
  Clipboard.setData(ClipboardData(text: logText));

  // Use navigatorContext if provided (from MaterialApp.builder), otherwise use widget context
  final navContext = widget.navigatorContext ?? context;

  // Show feedback using navigation callbacks
  final callbacks = SuperLogManager.navigationCallbacks;
  if (callbacks.onShowSnackbar != null) {
   callbacks.onShowSnackbar!(
    navContext,
    'All log messages copied to clipboard',
   );
  } else if (mounted) {
   // Fallback to ScaffoldMessenger
   ScaffoldMessenger.of(navContext).showSnackBar(
    SnackBar(
     content: const Text('All log messages copied to clipboard'),
     backgroundColor: Colors.green.withOpacity(0.9),
     duration: const Duration(seconds: 2),
    ),
   );
  }
 }
```

```
  void _showOverlay() {
    debugPrint(
      'SuperLogManager: _showOverlay called, isOverlayOpen: ${_isOverlayOpen.value}, mounted: $mounted',
    );
    if (_isOverlayOpen.value || !mounted) {
      debugPrint(
        'SuperLogManager: _showOverlay aborted - already open or not mounted',
      );
      return;
    }

    debugPrint('SuperLogManager: Setting _isOverlayOpen to true');
    _isOverlayOpen.value = true;

    // Navigate to full screen debug log using navigation callbacks
    // Add retry logic for when app is starting
    void tryNavigate() {
      debugPrint('SuperLogManager: tryNavigate called, mounted: $mounted');
      if (!mounted) {
        debugPrint('SuperLogManager: tryNavigate aborted - not mounted');
        return;
      }

      final callbacks = SuperLogManager.navigationCallbacks;
      debugPrint(
```

```
      'SuperLogManager: Navigation callbacks available: ${callbacks.onNavigateToDebugLog
!= null}',
    );

    // Use navigatorContext if provided (from MaterialApp.builder), otherwise use widget
context
    // Try widget context first (should have Navigator if bubble is in Navigator tree)
    // Then try navigatorContext (from builder)
    final navigatorFromKey = SuperLogManager.navigatorKey.currentState;
    final navigatorContextFromKey = navigatorFromKey?.context;

    BuildContext navContext = navigatorContextFromKey ?? context;

    // Check if widget context has Navigator access
    NavigatorState? testNavigator = Navigator.maybeOf(
      navContext,
      rootNavigator: true,
    );
    if (testNavigator == null &&
        widget.navigatorContext != null &&
        widget.navigatorContext != navigatorContextFromKey) {
      debugPrint(
        'SuperLogManager: Widget context has no Navigator, trying builder context',
      );
      navContext = widget.navigatorContext!;
      testNavigator = Navigator.maybeOf(navContext, rootNavigator: true);
    }
```

```
if (testNavigator == null && navigatorFromKey == null) {
  debugPrint('SuperLogManager: Neither context has Navigator access');
  debugPrint(
    'SuperLogManager: widget.context type: ${context.runtimeType}',
  );
  debugPrint(
    'SuperLogManager: navigatorContext type: ${widget.navigatorContext?.runtimeType}',
  );
}
final navigatorForPush = navigatorFromKey ?? testNavigator;

// If callbacks are provided (go_router, custom), use them
// Otherwise, use Navigator directly (MaterialApp.builder provides context)
if (callbacks.onNavigateToDebugLog != null) {
  debugPrint('SuperLogManager: Using navigation callback to navigate');
  callbacks
      .onNavigateToDebugLog!(
        navContext,
        () => const SuperDebugLogScreen(),
      )
      .then((_) {
        // Screen was closed
        if (mounted) {
          _isOverlayOpen.value = false;
        }
```

```
      })
      .catchError((e, stackTrace) {
        // Navigation failed, log and retry after delay
        debugPrint('SuperLogManager: Navigation callback error: $e');
        debugPrint('SuperLogManager: Stack trace: $stackTrace');
        if (mounted && _isOverlayOpen.value) {
          debugPrint(
            'SuperLogManager: Retrying navigation after delay...',
          );
          Future.delayed(const Duration(milliseconds: 500), () {
            if (mounted && _isOverlayOpen.value) {
              tryNavigate();
            }
          });
        } else {
          debugPrint(
            'SuperLogManager: Not retrying - not mounted or overlay closed',
          );
          _isOverlayOpen.value = false;
        }
      });
    } else {
      // No callbacks - use Navigator directly (MaterialApp.builder context)
      debugPrint(
        'SuperLogManager: Using Navigator directly (MaterialApp.builder)',
      );
```

```
if (navigatorForPush != null) {
  debugPrint('SuperLogManager: Found Navigator, pushing route');
  navigatorForPush
      .push(
        MaterialPageRoute(
          builder: (_) => const SuperDebugLogScreen(),
          fullscreenDialog: true,
        ),
      )
      .then((_) {
        if (mounted) {
          _isOverlayOpen.value = false;
        }
      })
      .catchError((e) {
        debugPrint('SuperLogManager: Navigation error: $e');
        if (mounted && _isOverlayOpen.value) {
          Future.delayed(const Duration(milliseconds: 500), () {
            if (mounted && _isOverlayOpen.value) {
              tryNavigate();
            }
          });
        }
      });
} else {
```

```
        debugPrint('SuperLogManager: Could not find Navigator');
        // Retry after delay
        if (mounted && _isOverlayOpen.value) {
          Future.delayed(const Duration(milliseconds: 500), () {
            if (mounted && _isOverlayOpen.value) {
              tryNavigate();
            }
          });
        }
      }
    }
  }

  // Wait for next frame to ensure context is ready
  debugPrint(
    'SuperLogManager: Scheduling post-frame callback for navigation',
  );
  WidgetsBinding.instance.addPostFrameCallback((_) {
    debugPrint(
      'SuperLogManager: Post-frame callback executed, mounted: $mounted,
isOverlayOpen: ${_isOverlayOpen.value}',
    );
    if (mounted && _isOverlayOpen.value) {
      debugPrint(
        'SuperLogManager: Calling tryNavigate from post-frame callback',
      );
```

```
      tryNavigate();
    } else {
      debugPrint(
        'SuperLogManager: Post-frame callback skipped - not mounted or overlay closed',
      );
    }
  });
 }

 void _closeOverlay() {
  if (!_isOverlayOpen.value || !mounted) return;

  // Use navigatorContext if provided (from MaterialApp.builder), otherwise use widget context
  final navContext = widget.navigatorContext ?? context;

  // Use navigation callbacks
  final callbacks = SuperLogManager.navigationCallbacks;
  if (callbacks.onNavigateBack != null) {
    callbacks.onNavigateBack!(navContext);
  } else {
    // Fallback to default Navigator
    final navigator = Navigator.of(navContext, rootNavigator: true);
    if (navigator.canPop()) {
      navigator.pop();
    }
```

```
  }
  _isOverlayOpen.value = false;
}

@override
Widget build(BuildContext context) {
  // Check if debug screen is open (optional feature)
  if (_config.hideBubbleWhenScreenOpen || widget.hideWhenScreenOpen) {
    final route = ModalRoute.of(context);
    if (route?.settings.name?.toLowerCase().contains('debug') == true) {
      return const SizedBox.shrink();
    }
  }

  // Get screen size for constraints
  final screenSize = MediaQuery.of(context).size;
  _screenSize = screenSize;

  // Determine text direction: check locale first, then fallback to Directionality
  final locale = Localizations.maybeLocaleOf(context);
  final newTextDirection = locale != null
      ? (locale.languageCode == 'ar' ? TextDirection.rtl : TextDirection.ltr)
      : Directionality.of(context);

  // Recalculate position if direction changed or position not initialized
  final needsRecalculation =
```

```
    _textDirection != newTextDirection || _position.value == Offset.zero;
_textDirection = newTextDirection;

// Initialize or recalculate position based on config
if (needsRecalculation) {
  WidgetsBinding.instance.addPostFrameCallback((_) {
    if (mounted) {
      final bubbleRadius = _bubbleSize / 2;
      final cornerPosition = _config.initialBubblePosition;

      // Calculate center position based on text direction:
      // - RTL: dx = distance from right edge (position on right)
      // - LTR: dx = distance from left edge (position on left)
      // - dy = distance from top edge (same for both)
      final centerX = _textDirection == TextDirection.rtl
          ? screenSize.width - cornerPosition.dx - bubbleRadius
          : cornerPosition.dx + bubbleRadius;
      final centerY = cornerPosition.dy + bubbleRadius;

      _position.value = Offset(centerX, centerY);
    }
  });
}

// Positioned must be a direct child of Stack (from DebugWrapper)
return ValueListenableBuilder<bool>(
```

```
valueListenable: _isOverlayOpen,
builder: (context, isOverlayOpen, child) {
 // Hide bubble if overlay is open
 if (isOverlayOpen ||
    ((_config.hideBubbleWhenScreenOpen || widget.hideWhenScreenOpen) &&
       _isOverlayOpen.value)) {
  return const SizedBox.shrink();
 }
 return ValueListenableBuilder<Offset>(
  valueListenable: _position,
  builder: (context, position, child) {
   final bubbleRadius = _bubbleSize / 2;
   return Positioned(
    left: position.dx - bubbleRadius,
    top: position.dy - bubbleRadius,
    child: Directionality(
     textDirection: TextDirection.ltr,
     child: GestureDetector(
      onPanStart: _config.enableBubbleDrag ? _onPanStart : null,
      onPanUpdate: _config.enableBubbleDrag
         ? (details) => _onPanUpdate(details, screenSize)
         : null,
      onPanEnd: _config.enableBubbleDrag ? _onPanEnd : null,
      onTap: _onTap,
      onLongPress: _onLongPress,
      child: Material(
```

```
color: Colors.transparent,
elevation: 8,
shape: const CircleBorder(),
child: Container(
 width: _bubbleSize,
 height: _bubbleSize,
 decoration: BoxDecoration(
  color: _config.bubbleColor,
  shape: BoxShape.circle,
  boxShadow: [
   BoxShadow(
    color: Colors.black.withOpacity(0.2),
    blurRadius: 8,
    offset: const Offset(0, 2),
   ),
  ],
 ),
 child: Stack(
  alignment: Alignment.center,
  children: [
   Icon(
    Icons.bug_report,
    color: _config.bubbleIconColor,
   ),
   // Error count badge
   ValueListenableBuilder<int>(
```

```
valueListenable: _errorCount,
builder: (context, count, child) {
 if (count == 0) return const SizedBox();
 return Positioned(
  top: 0,
  right: 0,
  child: Container(
   padding: const EdgeInsets.all(4),
   decoration: BoxDecoration(
    color: _config.errorBadgeColor,
    shape: BoxShape.circle,
   ),
   constraints: const BoxConstraints(
    minWidth: 18,
    minHeight: 18,
   ),
   child: Text(
    count > 99 ? '99+' : count.toString(),
    style: TextStyle(
     color: _config.errorBadgeTextColor,
     fontSize: 10,
     fontWeight: FontWeight.bold,
    ),
    textAlign: TextAlign.center,
   ),
  ),
```

```
                    );
                  },
                ),
              ],
            ),
          ),
        ),
      ),
    ),
  );
 }
}
import 'package:flutter/material.dart';
import 'debug_overlay_bubble.dart';
import 'debug_log_screen.dart';
import '../controllers/super_log_manager.dart';
import '../models/log_config.dart';

/// Widget to wrap the entire app and provide the debug overlay
///
/// For MaterialApp: Use SuperDebugWrapper.builder in MaterialApp.builder
/// For GetX: No special handling needed (GetX handles navigation)
```

```
/// For go_router: Use navigationCallbacks in SuperLogConfig
class SuperDebugWrapper extends StatelessWidget {
 final Widget child;

 const SuperDebugWrapper({super.key, required this.child});

 @override
 Widget build(BuildContext context) {
  // Check if LogManager is initialized and overlay is enabled
  final config = SuperLogManager.config;
  if (!SuperLogManager.isInitialized || config?.showOverlayBubble != true) {
   return child;
  }

  if (child is MaterialApp) {
   return _wrapMaterialApp(child as MaterialApp);
  }

  // For non-MaterialApp widgets (GetX, go_router, etc.), wrap with overlay host
  return Directionality(
   textDirection: TextDirection.ltr,
   child: _SuperDebugOverlayHost(child: child),
  );
 }

 Widget _wrapMaterialApp(MaterialApp materialApp) {
```

```
return MaterialApp(
 key: materialApp.key,
 navigatorKey: SuperLogManager.navigatorKey,
 home: materialApp.home,
 routes: materialApp.routes ?? const {},
 initialRoute: materialApp.initialRoute,
 onGenerateRoute: materialApp.onGenerateRoute,
 onGenerateInitialRoutes: materialApp.onGenerateInitialRoutes,
 onUnknownRoute: materialApp.onUnknownRoute,
 navigatorObservers: materialApp.navigatorObservers ?? [],
 builder: (context, child) {
  final builtChild =
     materialApp.builder?.call(context, child) ??
     child ??
     const SizedBox.shrink();
  return _SuperDebugOverlayHost(child: builtChild);
 },
 title: materialApp.title,
 debugShowMaterialGrid: materialApp.debugShowMaterialGrid,
 showPerformanceOverlay: materialApp.showPerformanceOverlay,
 checkerboardRasterCacheImages: materialApp.checkerboardRasterCacheImages,
 checkerboardOffscreenLayers: materialApp.checkerboardOffscreenLayers,
 showSemanticsDebugger: materialApp.showSemanticsDebugger,
 debugShowCheckedModeBanner: materialApp.debugShowCheckedModeBanner,
 theme: materialApp.theme,
 darkTheme: materialApp.darkTheme,
```

```
    themeMode: materialApp.themeMode,
    locale: materialApp.locale,
    localizationsDelegates: materialApp.localizationsDelegates,
    localeListResolutionCallback: materialApp.localeListResolutionCallback,
    localeResolutionCallback: materialApp.localeResolutionCallback,
    supportedLocales: materialApp.supportedLocales,
    restorationScopeId: materialApp.restorationScopeId,
    color: materialApp.color,
    themeAnimationDuration: materialApp.themeAnimationDuration,
    themeAnimationCurve: materialApp.themeAnimationCurve,
    scrollBehavior: materialApp.scrollBehavior,
    useInheritedMediaQuery: materialApp.useInheritedMediaQuery,
    shortcuts: materialApp.shortcuts,
    actions: materialApp.actions,
  );
}

/// Legacy builder still provided for backward compatibility
static Widget builder(BuildContext context, Widget? child) {
  if (!SuperLogManager.isInitialized ||
      SuperLogManager.config?.showOverlayBubble != true ||
      child == null) {
    return child ?? const SizedBox.shrink();
  }
  return _SuperDebugOverlayHost(child: child);
}
```

```
}

class _SuperDebugOverlayHost extends StatefulWidget {
 final Widget child;

 const _SuperDebugOverlayHost({required this.child});

 @override
 State<_SuperDebugOverlayHost> createState() => _SuperDebugOverlayHostState();
}

class _SuperDebugOverlayHostState extends State<_SuperDebugOverlayHost> {
 bool _isLogVisible = false;

 void _showLogs() {
  if (!_isLogVisible) {
   setState(() => _isLogVisible = true);
  }
 }

 void _hideLogs() {
  if (_isLogVisible) {
   setState(() => _isLogVisible = false);
  }
 }
```

```
 @override
 Widget build(BuildContext context) {
  final config = SuperLogManager.config ?? const SuperLogConfig();
  final bubbleContext =
     SuperLogManager.navigatorKey.currentContext ?? context;

  return Stack(
   children: [
    widget.child,
    if (!_isLogVisible || !config.hideBubbleWhenScreenOpen)
     SuperDebugOverlayBubble(
      onTap: _showLogs,
      hideWhenScreenOpen: config.hideBubbleWhenScreenOpen,
      navigatorContext: bubbleContext,
     ),
    if (_isLogVisible)
     _DebugLogOverlay(
      onClose: _hideLogs,
      dimBackground: config.dimOverlayBackground,
      heightFraction: config.panelHeightFraction,
     ),
   ],
  );
 }
}
```

```
/// Bottom panel overlay similar to debug_console_overlay
/// Uses Align instead of Positioned to avoid ParentDataWidget errors
class _DebugLogOverlay extends StatelessWidget {
 final VoidCallback onClose;
 final bool dimBackground;
 final double heightFraction;

 const _DebugLogOverlay({
  required this.onClose,
  this.dimBackground = true,
  this.heightFraction = 0.6,
 });

 @override
 Widget build(BuildContext context) {
  final media = MediaQuery.of(context);
  final screenHeight = media.size.height;
  final panelHeight = screenHeight * heightFraction;
  final overlayColor = dimBackground
     ? Colors.black.withOpacity(0.6)
     : Colors.transparent;

  // Build the panel content
  Widget panelContent = Container(
   height: panelHeight,
   width: media.size.width,
```

```
  decoration: BoxDecoration(
    color: Theme.of(context).scaffoldBackgroundColor,
    borderRadius: const BorderRadius.only(
      topLeft: Radius.circular(20),
      topRight: Radius.circular(20),
    ),
    boxShadow: [
      BoxShadow(
        color: Colors.black.withOpacity(0.2),
        blurRadius: 10,
        offset: const Offset(0, -2),
      ),
    ],
  ),
  child: ClipRRect(
    borderRadius: const BorderRadius.only(
      topLeft: Radius.circular(20),
      topRight: Radius.circular(20),
    ),
    child: SuperDebugLogScreen(onClose: onClose),
  ),
);

// Wrap with Localizations if needed
final hasMaterialLocalizations =
    Localizations.of<MaterialLocalizations>(
```

```
    context,
    MaterialLocalizations,
   ) !=
   null;
if (!hasMaterialLocalizations) {
 final locale =
    Localizations.maybeLocaleOf(context) ??
    WidgetsBinding.instance.platformDispatcher.locale;
 panelContent = Localizations(
  locale: locale,
  delegates: const [
   DefaultWidgetsLocalizations.delegate,
   DefaultMaterialLocalizations.delegate,
  ],
  child: panelContent,
 );
}

// Use Align instead of Positioned to avoid ParentDataWidget errors
return Stack(
 children: [
  // Dim background
  if (dimBackground)
   Positioned.fill(
    child: GestureDetector(
     onTap: onClose,
```

```
            child: Material(
              color: overlayColor,
              child: const SizedBox.expand(),
            ),
          ),
        ),
      // Bottom panel
      Align(alignment: Alignment.bottomCenter, child: panelContent),
    ],
  );
 }
}
import 'package:flutter/material.dart';
import 'package:flutter/services.dart';

/// Default error widget for SuperLogManager
/// Displays a user-friendly error screen when app initialization fails
class SuperDefaultErrorWidget extends StatelessWidget {
 final Object error;

 const SuperDefaultErrorWidget(this.error, {super.key});

 @override
 Widget build(BuildContext context) {
  final isDark = Theme.of(context).brightness == Brightness.dark;
  final errorMessage = error.toString();
```

```
return MaterialApp(
 theme: ThemeData.light(),
 darkTheme: ThemeData.dark(),
 themeMode: isDark ? ThemeMode.dark : ThemeMode.light,
 home: Scaffold(
  backgroundColor: isDark ? Colors.grey[900] : Colors.grey[100],
  appBar: AppBar(
   title: const Text('Application Error'),
   backgroundColor: Colors.red,
   foregroundColor: Colors.white,
  ),
  body: SafeArea(
   child: Center(
    child: SingleChildScrollView(
     padding: const EdgeInsets.all(24.0),
     child: Column(
      mainAxisAlignment: MainAxisAlignment.center,
      children: [
       Icon(Icons.error_outline, size: 80, color: Colors.red[400]),
       const SizedBox(height: 24),
       Text(
        'Failed to Initialize App',
        style: TextStyle(
         fontSize: 24,
         fontWeight: FontWeight.bold,
```

```
    color: isDark ? Colors.white : Colors.black87,
   ),
   textAlign: TextAlign.center,
  ),
  const SizedBox(height: 16),
  Container(
   padding: const EdgeInsets.all(16),
   decoration: BoxDecoration(
    color: isDark ? Colors.grey[800] : Colors.white,
    borderRadius: BorderRadius.circular(8),
    border: Border.all(
     color: Colors.red..withValues(alpha: 0.3),
     width: 1,
    ),
   ),
   child: SelectableText(
    errorMessage,
    style: TextStyle(
     fontSize: 14,
     color: isDark ? Colors.grey[300] : Colors.black87,
     fontFamily: 'monospace',
    ),
   ),
  ),
  const SizedBox(height: 24),
  ElevatedButton.icon(
```

```
              onPressed: () {
                SystemNavigator.pop();
              },
              icon: const Icon(Icons.exit_to_app),
              label: const Text('Exit App'),
              style: ElevatedButton.styleFrom(
                backgroundColor: Colors.red,
                foregroundColor: Colors.white,
                padding: const EdgeInsets.symmetric(
                  horizontal: 24,
                  vertical: 12,
                ),
              ),
            ),
          ],
        ),
      ),
    ),
  );
 }
}
/// A Flutter plugin for comprehensive debug logging and overlay management.
///
/// This package provides a complete debug logging solution with:
```

```
/// - Automatic error catching (Flutter errors, Dart errors, print/debugPrint interception)
/// - Draggable debug overlay bubble with error count badges
/// - Full-featured log viewer with search, filtering, and selection
/// - Performance optimizations and RTL/LTR support
/// - Customizable navigation callbacks for integration with different routing systems
///
/// ## Basic Usage
///
/// ```dart
/// import 'package:flutter_super_log_manager/flutter_super_log_manager.dart';
///
/// void main() {
///   // Run app with debug logging enabled
///   SuperLogManager.runApp(
///     MyApp(),
///     config: SuperLogConfig(
///       enabled: true,
///       showOverlayBubble: true,
///       capturePrint: true,
///       captureDebugPrint: true,
///     ),
///   );
/// }
/// ```
///
/// ## Advanced Usage
```

```
///
/// ```dart
/// void main() {
///   SuperLogManager.runApp(
///     MyApp(),
///     config: SuperLogConfig(
///       enabled: true,
///       maxLogs: 2000,
///       showOverlayBubble: true,
///       initialBubblePosition: Offset(32.0, 150.0),
///       bubbleColor: Colors.blue..withValues(alpha: 0.8),
///       enableLogSearch: true,
///       enableLogFiltering: true,
///       enableLogDeletion: true,
///       // Custom navigation for GetX, go_router, etc.
///       navigationCallbacks: SuperLogNavigationCallbacks(
///         onNavigateToDebugLog: (context, builder) async =>
///             await Get.toNamed('/debug-logs'),
///         onNavigateBack: (context) => Get.back(),
///         onShowSnackbar: (context, msg) => Get.snackbar('Info', msg),
///       ),
///     ),
///     preRun: () async {
///       // Initialize services before app starts
///       await initializeServices();
///       return true; // Return false to abort app start
```

```
///   },
///   postRun: () {
///     // Run after app is ready
///     initializeBackgroundTasks();
///   },
///  );
/// }
/// ```
///
/// ## Manual Logging
///
/// ```dart
/// // Add custom logs anywhere in your app
/// SuperLogManager.instance.addLog(
///  'User logged in successfully',
///  level: LogLevel.info,
///  tag: 'AUTH',
/// );
///
/// SuperLogManager.instance.addLog(
///  'Failed to load data',
///  level: LogLevel.error,
///  error: exception,
///  stackTrace: stackTrace,
/// );
/// ```
```

```
///
/// ## Disabling Debug Mode
///
/// ```dart
/// void main() {
///   // Completely disable debug logging
///   SuperLogManager.runApp(
///     MyApp(),
///     config: SuperLogConfig.disabled(),
///   );
/// }
/// ```
///
/// ## Custom Error Widget
///
/// ```dart
/// SuperLogManager.runApp(
///   MyApp(),
///   config: SuperLogConfig(),
///   errorWidget: (error) => CustomErrorScreen(error: error),
/// );
/// ```
library;

// Core functionality
export 'src/controllers/super_log_manager.dart';
```

```
export 'src/models/log_config.dart';
export 'src/models/log_navigation_callbacks.dart';
export 'src/views/default_error_widget.dart';

// Data models
export 'src/models/log_entry.dart';

// UI components
export 'src/views/debug_wrapper.dart';
export 'src/views/debug_overlay_bubble.dart';
export 'src/views/debug_log_screen.dart';
```