### Refactoring Patterns 1: Extract Method (firebase\_service.dart)

Future<List<TimeEntry>> getAllTasks() async {

try {

final firestore = FirebaseFirestore.instance; // Initializing Firestore here

final snapshot = await firestore.collection('time\_entries').get();

print('Fetched documents: ${snapshot.docs.length}'); // Verbose logging

return snapshot.docs.map((doc) {

final data = doc.data();

return TimeEntry(

id: doc.id,

task: data['task'],

date: data['date'],

from: data['from'],

to: data['to'],

tag: data['tag'],

); // Mapping logic inline

}).toList();

} catch (e) {

print('Error fetching tasks: $e');

throw Exception('Failed to fetch tasks');

}

}

### Refactored Code

Future<List<TimeEntry>> getAllTasks() async {

try {

final snapshot = await \_db.collection('time\_entries').get();

print('Fetched documents: ${snapshot.docs.length}'); // Debug

return snapshot.docs.map((doc) {

return TimeEntry.fromMap(doc.data(), doc.id); // Pass document ID

}).toList();

} catch (e) {

print('Error fetching tasks: $e');

throw Exception('Failed to fetch tasks');

}

}

**Explanation of Refactor:**

Smell: Firestore instance is initialized inline repeatedly instead of reusing a single instance.

Solution: Extracted Firestore instance to a class-level variable \_db for reuse.

Smell: Mapping logic is inline, making the code harder to maintain and read.

Solution: Moved mapping logic to TimeEntry.fromMap, centralizing object creation.

### Refactoring Patterns 2: Rename for Clarity (firebase\_service.dart)

Future<void> saveTaskToDatabase(TimeEntry task) async {

try {

await \_db.collection('time\_entries').add({

'task': task.task,

'date': task.date,

'from': task.from,

'to': task.to,

'tag': task.tag,

});

print('Saved new task to database.');

} catch (e) {

print('Failed to save task: $e');

throw Exception('Database save operation failed');

}

}

### Refactored Code

Future<void> addTask(TimeEntry task) async {

try {

await \_db.collection('time\_entries').add(task.toMap());

print('Task added successfully');

} catch (e) {

print('Error adding task: $e');

throw Exception('Failed to add task');

}

}

**Explanation of Refactor:**

Smell: Method name saveTaskToDatabase is verbose and inconsistent with naming conventions.

Solution: Renamed to addTask for brevity and clarity.

Smell: Object conversion logic is inline, making the code repetitive and hard to maintain.

Solution: Used task.toMap() to encapsulate the conversion logic within the TimeEntry model.

### Refactoring Patterns 3: Extract Method (query\_report\_screen.dart)

void \_searchTasks(String query) async {

setState(() => \_isLoading = true);

try {

List<TimeEntry> allTasks = await \_firebaseService.getAllTasks();

List<TimeEntry> results = allTasks.where((task) {

final lowerQuery = query.toLowerCase();

return task.task.toLowerCase().contains(lowerQuery) ||

task.tag.toLowerCase().contains(lowerQuery) ||

task.date.contains(query);

}).toList();

results = results..sort((a, b) => DateTime.parse(a.date).compareTo(DateTime.parse(b.date)));

setState(() {

\_searchResults = results;

\_tagReport = {};

});

} catch (e) {

print('Error searching tasks: $e');

} finally {

setState(() => \_isLoading = false);

}

}

### Refactored Code

void \_searchTasks(String query) async {

setState(() => \_isLoading = true);

try {

List<TimeEntry> allTasks = await \_firebaseService.getAllTasks();

// Filter tasks by query

List<TimeEntry> results = allTasks.where((task) {

final lowerQuery = query.toLowerCase();

return task.task.toLowerCase().contains(lowerQuery) ||

task.tag.toLowerCase().contains(lowerQuery) ||

task.date.contains(query);

}).toList();

// Sort results by date and time

results = sortTasksByDateTime(results);

setState(() {

\_searchResults = results;

\_tagReport = {}; // Clear report view when searching

});

} catch (e) {

print('Error searching tasks: $e');

} finally {

setState(() => \_isLoading = false);

}

}

**Explanation of Refactor:**

Smell: Inline sorting logic cluttered the method and made it less reusable.

Solution: Extracted sorting into sortTasksByDateTime, improving code clarity and reusability.

### Refactoring Patterns 4: Replace Temp with Query (query\_report\_screen.dart)

void \_generateReport() async {

setState(() => \_isLoading = true);

try {

List<TimeEntry> allTasks = await \_firebaseService.getAllTasks();

Map<String, Duration> report = calculateTimeByTag(allTasks);

final sortedReport = Map.fromEntries(

report.entries.toList()..sort((a, b) => b.value.compareTo(a.value)),

);

setState(() {

\_tagReport = sortedReport;

\_searchResults = [];

});

} catch (e) {

print('Error generating report: $e');

} finally {

setState(() => \_isLoading = false);

}

}

### Refactored Code

void \_generateReport() async {

setState(() => \_isLoading = true);

try {

List<TimeEntry> allTasks = await \_firebaseService.getAllTasks();

Map<String, Duration> report = calculateTimeByTag(allTasks);

final sortedReport = Map.fromEntries(

report.entries.toList()..sort((a, b) => b.value.compareTo(a.value)),

);

setState(() {

\_tagReport = sortedReport;

\_searchResults = []; // Clear search results when generating report

});

} catch (e) {

print('Error generating report: $e');

} finally {

setState(() => \_isLoading = false);

}

}

**Explanation of Refactor:**

Smell: The temporary variable report.entries.toList() added unnecessary overhead.

Solution: Replaced the temp variable by directly using .entries.toList() in the Map.fromEntries() call. This simplifies the code and improves readability.

### Refactoring Patterns 5: Decompose Conditional (query\_report\_screen.dart)

List<TimeEntry> results = allTasks.where((task) {

try {

final taskDate = DateFormat('yyyy/MM/dd').parse(task.date);

if (startDate != null && taskDate.isBefore(startDate) && !taskDate.isAtSameMomentAs(startDate)) {

return false;

}

if (endDate != null && taskDate.isAfter(endDate) && !taskDate.isAtSameMomentAs(endDate)) {

return false;

}

return true;

} catch (e) {

print('Error parsing task date: $e');

return false;

}

}).toList();

### Refactored Code

List<TimeEntry> results = allTasks.where((task) {

try {

DateTime taskDate = dateFormat.parse(task.date);

return (startDate == null || taskDate.isAfter(startDate) || taskDate.isAtSameMomentAs(startDate)) &&

(endDate == null || taskDate.isBefore(endDate) || taskDate.isAtSameMomentAs(endDate));

} catch (e) {

print('Error parsing task date: $e');

return false;

}

}).toList();

**Explanation of Refactor:**

Smell: The old code uses redundant if statements for checking the date range, leading to verbosity and unnecessary complexity.

Solution: Replaced the conditional blocks with a concise, single return statement using logical operators (|| and &&). This reduces complexity and improves readability without changing functionality.