Code Smell 1 (home_screens.dart)

_upcomingTasks = tasks.sublist(0, tasks.length < 5 ? tasks.length : 5); // Manual slicing logic

Refactored Code

_upcomingTasks = tasks.take(5).toList();

Refactoring Explanation:
Code Smell: Manual boundary checks and slicing logic are harder to read and error more often.

Solution: Replaced with .take(5).toList() for clarity and improved readability.

Code Smell 2 (home_screens.dart)

Text(task.task), // No text style

Refactored Code

Text(task.task, style: TextStyle(color: Colors.white)),

Refactoring Explanation:
Code Smell: Missing consistent styling for text elements.
Solution: Added a TextStyle to align with the app's theme, improving readability and visual coherence.

Code Smell 3 (home_screens.dart)

print('Error occurred: ' + e.toString()); // String concatenation for logging

Refactored Code

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

Refactoring Explanation:
Code Smell: String concatenation in logging is verbose and less readable.
Solution: Used string interpolation ('$e') to simplify and improve log readability.

Code Smell 4 (home_screens.dart)

tasks.sort((a, b) => a.date.compareTo(b.date)); // Inline sorting

Refactored Code

```
tasks = sortTasksByDateTime(tasks);
```

Refactoring Explanation:
Code Smell: Sorting logic inline leads to duplication if used elsewhere.
Solution: Moved sorting to a utility function for better reusability and separation of concerns.

Code Smell 5 (task_utils.dart)

```
tasks.sort((a, b) {
  return DateFormat('yyyy/MM/dd h:mm a').parse('${a.date} ${a.from}')
      .compareTo(DateFormat('yyyy/MM/dd h:mm a').parse('${b.date} ${b.from}'));
}); // Repeated inline date parsing
```

Refactored Code

```
final dateFormat = DateFormat('yyyy/MM/dd h:mm a'); // Define the custom format

tasks.sort((a, b) {
  DateTime dateTimeA = dateFormat.parse('${a.date} ${a.from}');
  DateTime dateTimeB = dateFormat.parse('${b.date} ${b.from}');
  return dateTimeA.compareTo(dateTimeB);
});
```

Refactoring Explanation:
Code Smell: Repeated initialization of DateFormat inline in the sort logic.
Solution: Extracted DateFormat initialization to a single variable to avoid redundancy and improve performance.

Code Smell 6 (task_utils.dart)

```
if (query == '' || query == null) return tasks; // Overly verbose null/empty check
```

Refactored Code

```
if (query.isEmpty) return tasks;
```

Refactoring Explanation:
Code Smell: Verbose null/empty string check (query == '' || query == null).
Solution: Replaced with the more concise and readable query.isEmpty check.

Code Smell 7 (task_utils.dart)

```
print('Error parsing task dates: ' + e.toString()); // String concatenation in logging
```

Refactored Code

```
print('Error parsing task dates: $e');
```

Refactoring Explanation:
Code Smell: String concatenation for error logging is verbose and harder to read.
Solution: Replaced with string interpolation ('$e') for simplicity and cleaner log output.

Code Smell 8: Long Methods (firebase_service.dart)

```
Future<List<TimeEntry>> getAllTasks() async {
 final snapshot = await FirebaseFirestore.instance.collection('time_entries').get();
 print('Fetched documents: ' + snapshot.docs.length.toString()); // Inline logging
 List<TimeEntry> tasks = [];
 for (var doc in snapshot.docs) {
  tasks.add(TimeEntry.fromMap(doc.data(), doc.id)); // Mapping inline
 }
 return tasks;
}
```

Refactored Code

```
final _db = FirebaseFirestore.instance;

Future<List<TimeEntry>> getAllTasks() async {
 try {
  final snapshot = await _db.collection('time_entries').get();
  print('Fetched documents: ${snapshot.docs.length}'); // Cleaner logging
  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');
 }
}
```

Refactoring Explanation:
Smell: The old method was too long, performing Firestore calls, logging, and inline mapping in one place.

Solution: Separated Firestore initialization into _db, cleaned up logging with string interpolation, and used .map() to replace inline mapping, improving readability.

Code Smell 9: Feature Envy (firebase_service.dart)

```
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 every field inline
}).toList();
```

Refactored Code

```
return snapshot.docs.map((doc) {
  return TimeEntry.fromMap(doc.data(), doc.id); // Pass document ID
}).toList();
```

Refactoring Explanation:
Smell: The old code manually maps each field from doc.data(), cluttering the service and creating duplication risk.
Solution: Delegated mapping logic to TimeEntry.fromMap, simplifying service code and centralizing object creation in the TimeEntry model.

Code Smell 10: Inconsistent Naming (firebase_service.dart)

```
Future<void> AddTask(TimeEntry task) async {
  try {
    await _db.collection('time_entries').add(task.toMap());
    print('Task successfully added.');
  } catch (err) {
    print('Error adding task: ' + err.toString());
  }
}
```

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');
  }
}
```

Refactoring Explanation:
Smell: The old method uses inconsistent casing for the method name (AddTask instead of addTask) and inconsistent error logging.
Solution: Standardized method naming to camelCase and improved logging with string interpolation.

Code Smell 11: Primitive Obsession (firebase_service.dart)

```
await _db.collection('time_entries').doc(taskId).delete(); // Directly passing taskId as a string
```

Refactored Code

```
Future<void> deleteTask(String taskId) async {
  try {
    await _db.collection('time_entries').doc(taskId).delete();
    print('Task $taskId deleted successfully.');
  } catch (e) {
    print('Error deleting task: $e');
  }
}
```

Refactoring Explanation:
Smell: Passing taskId as a raw string makes the API harder to extend or validate.
Solution: Encapsulate the task ID in a domain-specific TaskIdentifier object if further validation or operations are needed in the future.