JSON File

Read and Write

Sync File Operations

```
import 'dart:io';
final content = file.readAsStringSync();
file.writeAsStringSync(content);
directory.existsSync()
directory.createSync(recursive: true);
```

- We must use sync functions for sync file operations.

SYNC File Read

```dart
import 'dart:io';

String? loadConfigSync(String filePath) {
  try {
    final file = File(filePath);
    if (!file.existsSync()) {
      throw FileSystemException('File not found', filePath);
    }
    // This blocks until complete!
    final content = file.readAsStringSync();
    print('File content loaded successfully:\n$content');
    return content;
  } catch (e) {
    print('Error loading file: $e');
    return null;
  }
}
```

Sync File Write

```
bool saveConfigSync(String filePath, String content) {
  try {
    final file = File(filePath);
    final directory = file.parent;
    if (!directory.existsSync()) {
      directory.createSync(recursive: true);
    }
    // This blocks until the write operation is complete!
    file.writeAsStringSync(content);
    return true;
  } catch (e) {
    print('Error writing file: $e');
    return false;
  }
}
```

Return value types

- For loadConfigSync, we return a String that is read from the function.
  - However, when an error occurs, we return null.
  - So, the return type is String?.
- For saveConfigSync, we return true/false only.
  - So, the return type is bool.

Async File Operations

```
import 'dart:io';
final content = await file.readAsString();
await file.writeAsString(content);
```

- Use async files for file operations.

```
await directory.exists()
await directory.create(recursive: true);
```

Async File Read

```
// ASYNC METHOD: Non-blocking file read
Future<String?> loadConfigAsync(String filePath) async {
  try {
    print('Reading file asynchronously...');

    final file = File(filePath);
    if (!await file.exists()) {
      throw FileSystemException('File not found', filePath);
    }
    // This doesn't block other operations!
    final content = await file.readAsString();
    print('File content loaded successfully:\n$content');
    return content;
  } catch (e) {
    print('Error: $e');
    return null;
    // rethrow;
  }
}
```

Async File Write

```
Future<bool> saveConfigAsync(String filePath, String content) async {
  try {
    print('Writing file asynchronously...');
    final file = File(filePath);
    final directory = file.parent;
    if (!await directory.exists()) {
      await directory.create(recursive: true);
    }
    // This doesn't block other operations!
    await file.writeAsString(content);
    return true;
  } catch (e) {
    print('Error writing file: $e');
    return false;
  }
}
```

When to Use Async vs Sync?

- Use **Sync** When:
  - Simple, quick operations
  - CPU-intensive calculations
  - When you need a guaranteed order
  - Educational examples (easier to understand)

- Use **Async** When:
  - Reading/writing files
  - Network requests
  - Database operations
  - Any I/O that might take time
  - You want responsive applications

Exception Handling

Without Exception Handling:

```
// Dangerous! Can crash your app
final content = await file.readAsString();
final data = jsonDecode(content);
```

- File I/O Applications Need Error Handling

With Exception Handling:

```
try {
  final content = await file.readAsString();
  final data = jsonDecode(content);
} on FileSystemException catch (e) {
  print('File error: ${e.message}');
} on FormatException catch (e) {
  print('JSON error: ${e.message}');
} catch (e) {
  print('Unexpected error: $e');
}
```

Exception Handling Best Practices

1. **Be Specific**: Catch specific exception types first
2. **Provide Recovery**: Create default files when possible
3. **Log Meaningful Messages**: Help with debugging
4. **Don't Ignore Errors**: Always handle or log them

```
try {
  // risky operation
} on SpecificException catch (e) {
  // handle specific case
} catch (e) {
  // handle general case
} finally {
  // cleanup (runs always)
}
```

5. **Use Finally When Needed**: Cleanup resources

Exception Handling Propagation

```
Future<String?> loadConfigAsync(String filePath) async {
  try {
    ...
    return content;
  } catch (e) {
    return null;
  }
}
```

- The loadConfigAsync uses functions that throw exceptions, so we use try/catch.

```
if (!await file.exists()) {
  throw FileSystemException('File not found', FilePath);
}
```

- When an error occurs, we rethrow exceptions.
- So, when we use the function, we should use try/catch again.

```
try {
  String result = await loadConfigAsync(configPath);
  print('Loading configuration from $configPath: content $result');
} catch (e) {
  print('Error: $e');
}
```

Stop the propagation of the exception

```
Future<bool> saveConfigAsync(String filePath, String content) async {
  try {
    ...
    return true;
  } catch (e) {
    ...
    return false;
  }
}
```

- The saveConfigAsync does not throw any errors, but returns true/false.
- So the functions that use it check the return value.

JSON: Data Serialization

- JSON (JavaScript Object Notation) is a widely adopted format for storing and exchanging information between systems.

```
{
  "foo":"hello",
  "bar":32
}
```

- The majority of API servers rely on JSON to structure and transmit data during communication with clients and other services.
- We need to use JSON as an intermediate format for storing data class information.

JSON Format

```
{
  "data": {
    "data": {
      "foo": "value",
      "bar": 123
    }
  }
}
```

- The JSON object is enclosed in an object {...} or an array [...].

```
"foo": "value"
"data": {...}
```

- Each element has the key: value dictionary structure.
  - The value can be any object or primitive value.
  - The key is called "Property Name".

JSON representation in Dart Language

- We use a Map<String, dynamic> data structure to represent a JSON object.
- We use jsonEncode to make a JSON string from the data structure.
- We use jsonDecode to make the data structure from a JSON string.

```dart
// ✅ Both are valid in Dart
Map<String, dynamic> option1 = {
  'foo': 'value', 'bar': 123
};
Map<String, dynamic> option2 = {
  "foo": "value", "bar": 123
};
```

- We can use both '...' and "..." for strings.

```
{"foo":"value", "bar":123}
```

- We should use only "..." in JSON strings.

Dart String Convention

- **Prefer** single quotes (`'`) by default
- **Use** double quotes (`"`) when a string contains apostrophes

JSON string

- Always use (`"`).

Comparison: JavaScript

```javascript
// ✅ Both valid in JavaScript
const obj1 = {
  'data': {
    'data': { 'foo': 'value', 'bar': 123 }
  }
};
const obj2 = {
  "data": {
    "data": { "foo": "value", "bar": 123 }
  }
};
```

You DON'T Use Quotes Property Names (keys) in JavaScript

```
// ✅ Simple property names - no quotes needed
const obj = {
  data: {                // No quotes!
    foo: 'value',        // Property name: no quotes
    bar: 123             // Value: number, no quotes
  }
};

// ⚠️ Special cases - quotes needed
const obj2 = {
  'my-property': 'value',       // Hyphens need quotes
  'property with space': 42,    // Spaces need quotes
  '123numeric': 'value'         // Starting with number
};
```

Storing and Retrieving JSON Strings

jsonRead

```
Future<Map<String, dynamic>?> jsonRead(String jsonPath) async {
  try {
    String result = await loadConfigAsync(jsonPath) ?? '';
    Map<String, dynamic> dartMap = jsonDecode(result);
    return dartMap;
  } catch (e) {
    return null;
  }
}
```

```dart
import 'dart:convert';
String result = await loadConfigAsync(jsonPath) ?? '';
```

- Step 1: Read file content into a string.
  - The loadConfigAsync returns String?, so we should make it String.

- Step 2: Convert the string into `Map<String, dynamic>`.
  - Return the converted Dart map.

```
Map<String, dynamic> dartMap = jsonDecode(result);
return dartMap;
```

Usage

```
Map<String, dynamic>? readMap = await jsonRead(jsonPath);
if (readMap != null) {
  print('JSON read successfully: $readMap');
} else {
  print('Failed to read JSON or JSON is empty.');
}
```

- jsonRead function returns Map<String, dynamic>? type, but it does not throw exceptions.
- So, we only check the nullity.

jsonWrite

```
Future<bool> jsonWrite(
  String jsonPath, Map<String, dynamic> dartMap) async {
  String jsonString = jsonEncode(dartMap);
  try {
    bool success = await saveConfigAsync(jsonPath, jsonString);
    if (success) {
      return true;
    } else {
      return false;
    }
  } catch (e) {
    return false;
  }
}
```

```dart
import 'dart:convert';
String jsonString = jsonEncode(dartMap);
```

- Step 1: Convert the Dart map into a JSON string.

```dart
bool success =
  await saveConfigAsync(jsonPath, jsonString);
```

- Step 2: Save the JSON string.

Dart object to JSON

- We need to convert any Dart class (object) into JSON to use for the REST API or store in files.
- We need to retrieve files in JSON string format into a Dart class.
- For this goal, Dart classes need to support toJson and fromJson functions.

We have a Data model: MarpConfig

```dart
class MarpConfig {
  final String theme;
  final String size;
  final bool math;
  final String author;

  // Convert object to JSON map
  Map<String, dynamic> toJson() {
    return {
      'theme': theme, 'size': size,
      'math': math,   'author': author,
    };
  }
```

- Add toJson method to convert Dart object into JSON.

Add factory to make Dart object from JSON

```dart
  // Create object from JSON map
  factory MarpConfig.fromJson(Map<String, dynamic> json) {
    return MarpConfig(
      theme: json['theme'] ?? 'default',
      size: json['size'] ?? '16:9',
      math: json['math'] ?? false,
      author: json['author'] ?? 'Anonymous',
    );
  }
}
```

We can transform a Dart object into a JSON string and vice versa.

```
var m = MarpConfig(...);
var jsonString = jsonEncode(m.toJson());
print(jsonString);
// {"theme":"default","size":"16:9",
//  "math":false,"author":"Anonymous"}

var marp = jsonDecode(jsonString);
var m2 = MarpConfig.fromJson(marp);
print(m2); // Instance of 'MarpConfig'
print(m2.theme); // default
```