Lab: Libraries in Dart Using Cloud IDE

Estimated time: **30** minutes

Welcome to this hands-on lab, where you will practice using various libraries in Dart, both built-in and user-defined. Libraries help organize your code, manage dependencies, and reuse components across different parts of your application.

Objectives

After completing this lab, you will be able to:

- Use common built-in Dart libraries such as `dart:core`, `dart:math`, `dart:async`, and `dart:convert`
- Install and use external Dart packages such as `http`, `intl`, and `path`
- Create your own custom libraries in Dart
- Import and use these libraries in your Dart applications

About Cloud IDE

Running the lab

This lab is designed to be completed using a Cloud IDE environment. You will write and execute Dart code directly within the IDE.

About Skills Network Cloud IDE

Skills Network Cloud IDE (based on Theia and Docker) provides an environment for hands-on labs for course and project-related labs. It is an open-source IDE (Integrated Development Environment).

Important notice about this lab environment

Please be aware that sessions for this lab environment are not persistent. Every time you connect to this lab, a new environment is created for you. Any data you may have saved in the earlier session will be lost. Plan to complete these labs in a single session to avoid losing your data.

Key terms

- **Library**: A collection of code (classes, functions, variables) that can be reused.
- `dart:core`: The core library is automatically imported into every Dart program.
- `dart:math`: A library that provides mathematical constants and functions.
- `dart:async`: A library for asynchronous programming.
- `dart:convert`: A library for encoding and decoding data.
- `http`: A package for making HTTP requests.
- `intl`: A package for internationalization and localization.
- `path`: A package for manipulating file system paths.

Step 1: Prerequisites

Let's create a `pubspec.yaml` file that holds configuration for your application.

1. Create a folder for your project called `dart-libs`.

   ```
   mkdir /home/project/dart-libs
   ```

   Change into the `dart-libs` folder.

   ```
   cd /home/project/dart-libs
   ```

2. Create a `pubspec.yaml` file in the project folder:

```
touch /home/project/dart-libs/pubspec.yaml
```

3. Open the `pubspec.yaml` file in the IDE.

Open **pubspec.yaml** in IDE

4. Add the following dependencies under the `dependencies` section:

```
name: dart_libs
description: >-
  Lab to practice with Dart libraries
environment:
  sdk: '^3.5.0'
dependencies:
  http: ^1.2.2
  intl: ^0.19.0
  path: ^1.9.0
dev_dependencies:
  test: '>=1.15.0 <2.0.0'
```

5. Install the dependencies by running the following command in your terminal:

```
dart pub get
```

The output should return with a success message:

```
Resolving dependencies... (1.1s)
Downloading packages... (4.6s)
+ _fe_analyzer_shared 73.0.0 (74.0.0 available)
+ _macros 0.3.2 from sdk dart
+ analyzer 6.8.0 (6.9.0 available)
+ args 2.5.0
...
...
+ webkit_inspection_protocol 1.2.1
+ yaml 3.1.2
Changed 52 dependencies!
3 packages have newer versions incompatible with dependency constraints.
```

This will install the required libraries for your project.

Step 2: Use the dart:core library

1. Create core.dart in the `/home/project/dart-libs` folder.

```
touch /home/project/dart-libs/core.dart
```

Open the file:

Open **core.dart** in IDE

2. The `dart:core` library is automatically imported into every Dart program and provides fundamental classes such as `String`, `int`, `List`, and `Map`.

```
void main() {
  String greeting = 'Hello, Dart!';
  int number = 42;
  List<String> fruits = ['Apple', 'Banana', 'Cherry'];
  Map<String, int> scores = {'Alice': 90, 'Bob': 85};
  print(greeting);
  print(number);
  print(fruits);
  print(scores);
}
```

   - The code declares and initializes variables using `dart:core` types.

3. Run your code to see the output:

```
dart /home/project/dart-libs/core.dart
```

The output should look like this:

```
Hello, Dart!
42
[Apple, Banana, Cherry]
{Alice: 90, Bob: 85}
```

Step 3: Use the dart:math library

1. Create `math.dart` in the `/home/project/dart-libs` folder.

```
touch /home/project/dart-libs/math.dart
```

Open the file:

Open **math.dart** in IDE

2. The `dart:math` library provides mathematical functions and constants.

```
import 'dart:math';
void main() {
  double angle = pi / 4;
  double sine = sin(angle);
  double cosine = cos(angle);
  Random random = Random();
  int randomNumber = random.nextInt(100);
  print('Sine: $sine');
  print('Cosine: $cosine');
  print('Random Number: $randomNumber');
}
```

   - The code uses `dart:math` to perform mathematical calculations and generate random numbers.

3. Run your code to see the output:

```
dart /home/project/dart-libs/math.dart
```

The output should look like this:

```
Sine: 0.7071067811865475
Cosine: 0.7071067811865475
Random Number: [a number between 0 and 99]
```

Step 4: Use the dart:async library

1. Create async.dart in `/home/project/dart-libs` folder.

```
touch /home/project/dart-libs/async.dart
```

Open the file:

Open **async.dart** in IDE

2. The `dart:async` library supports asynchronous programming with Futures and Streams.

```
import 'dart:async';
void main() async {
  Future<String> fetchData() async {
    await Future.delayed(Duration(seconds: 2));
    return 'Data fetched!';
  }
  String data = await fetchData();
  print(data);
}
```

- The code demonstrates asynchronous programming using `dart:async`.

3. Run your code to see the output after a delay of 2 seconds:

```
dart /home/project/dart-libs/async.dart
```

The output should look like this:

```
Data fetched!
```

Step 5: Use the dart:convert library

1. Create convert.dart in the `/home/project/dart-libs` folder.

```
touch /home/project/dart-libs/convert.dart
```

Open the file:

Open **convert.dart** in IDE

2. The `dart:convert` library provides utilities for encoding and decoding JSON.

```
import 'dart:convert';
void main() {
  String jsonString = '{"name": "Alice", "age": 30}';
  Map<String, dynamic> user = jsonDecode(jsonString);
  print('Name: ${user['name']}');
  print('Age: ${user['age']}');
  Map<String, dynamic> newUser = {'name': 'Bob', 'age': 25};
  String newJsonString = jsonEncode(newUser);
  print(newJsonString);
}
```

- The code demonstrates JSON encoding and decoding using `dart:convert`.

3. Run your code to see the output:

```
dart /home/project/dart-libs/convert.dart
```

The output should look like this:

```
Name: Alice
Age: 30
{"name":"Bob","age":25}
```

Step 6: Use the http package

1. Create http.dart in the `/home/project/dart-libs` folder.

```
touch /home/project/dart-libs/http.dart
```

Open the file:

Open **http.dart** in IDE

2. The `http` package is used for making HTTP requests.

```
import 'package:http/http.dart' as http;
void main() async {
  var url = Uri.parse('https://jsonplaceholder.typicode.com/posts/1');
  var response = await http.get(url);
  if (response.statusCode == 200) {
    print('Response data: ${response.body}');
  } else {
    print('Request failed with status: ${response.statusCode}');
  }
}
```

- The code performs a GET request to a placeholder API using the `http` package.

3. Run your code to see the output:

```
dart /home/project/dart-libs/http.dart
```

The output should look like this:

```
Response data: {
  "userId": 1,
  "id": 1,
  "title": "sunt aut facere repellat provident occaecati excepturi optio reprehenderit",
  "body": "quia et suscipit\nsuscipit recusandae consequuntur expedita et cum\nreprehenderit molestiae ut ut quas totam\nnostrum rerum est autem
}
```

Step 7: Use the intl package

1. Create intl.dart in the `/home/project/dart-libs` folder.

```
touch /home/project/dart-libs/intl.dart
```

Open the file:

Open **intl.dart** in IDE

2. The `intl` package is used for internationalization and localization.

```
import 'package:intl/intl.dart';
void main() {
  var now = DateTime.now();
  var formatter = DateFormat('yyyy-MM-dd');
  String formattedDate = formatter.format(now);
  print('Formatted date: $formattedDate');
}
```

   - The code demonstrates date formatting using the `intl` package.

3. Run your code to see the output:

```
dart /home/project/dart-libs/intl.dart
```

The output should look like this:

```
Formatted date: 2024-09-06
```

Step 8: Use the path package

1. Create path.dart in the `/home/project/dart-libs` folder.

```
touch /home/project/dart-libs/path.dart
```

Open the file:

Open **path.dart** in IDE

2. The `path` package provides functions for manipulating file system paths.

```
import 'package:path/path.dart' as p;
void main() {
  var fullPath = p.join('directory', 'file.txt');
  print('Full path: $fullPath');
}
```

- The code joins directory and file names to create a full path.

3. Run your code to see the output:

```
dart /home/project/dart-libs/path.dart
```

The output should look like this:

```
Full path: directory/file.txt
```

Step 9: Create and use your own library

1. Create custom.dart in the `/home/project/dart-libs` folder.

```
touch /home/project/dart-libs/custom.dart
```

Open the file:

Open **custom.dart** in IDE

2. Add the following code to define your library:

```
// custom.dart
library math_utils;
int add(int a, int b) {
  return a + b;
}
int subtract(int a, int b) {
  return a - b;
}
```

3. Create main.dart in the `/home/project/dart-libs` folder.

```
touch /home/project/dart-libs/main.dart
```

Open the file:

Open **main.dart** in IDE

- In your `main.dart` file, import the library and use it:

```
import 'custom.dart';
void main() {
  int sum = add(10, 5);
  int difference = subtract(10, 5);
  print('Sum: $sum');
  print('Difference: $difference');
}
```

4. Run your code to see the output:

```
dart /home/project/dart-libs/main.dart
```

The output should look like this:

```
Sum: 15
Difference: 5
```

Conclusion and next steps

Congratulations on completing this lab! You have now practiced using various Dart libraries, both built-in and custom, to enhance your Dart applications, manage dependencies, and organize your code more efficiently. You've explored essential libraries such as dart:core, dart:math, dart:async, and dart:convert, as well as external packages like http, intl, and path. Additionally, you have gained experience creating and importing custom libraries, further boosting your ability to structure and scale your Dart projects.

Now that you've completed the lab on using various Dart libraries, here are a few suggested next steps to solidify further the concepts covered in this lab:

1. Explore More Built-In Dart Libraries
    - Dart offers many more built-in libraries, such as dart:io for file and network I/O and dart:ffi for foreign function interfaces. Continue exploring these libraries and see how they can be applied to your projects.
    - **Example**: Try using dart:io to read and write files from your local filesystem.
2. Work with More External Packages
    - Beyond the packages explored in this lab, Dart has an extensive ecosystem of external packages available on pub.dev. Explore additional useful packages such as provider for state management or equatable for value comparison.
    - **Example**: Use provider to implement state management in your application, sharing data across different parts of your app.
3. Create Advanced Custom Libraries
    - You have already created a simple custom library. As your projects grow, you will need more complex libraries. Practice organizing your code into multiple libraries for better modularity and reusability.
    - **Example**: Create a library that handles various mathematical functions, string utilities, or API interaction, and import it into multiple Dart projects.

Author(s)

Skills Network