Lab: Explore Plugins in Flutter

Estimated time: **30 minutes**

Welcome to this hands-on lab where you will explore and utilize plugins in Flutter to enhance your app's functionality. This lab will help you understand how to find, add, and use plugins from the official Dart package repository.

Objectives

After completing this lab, you will be able to:

- Find and select plugins from the Dart package repository (pub.dev)
- Add plugins to a Flutter project and configure them
- Implement plugins to utilize native functionalities, such as launching URLs and storing data

About Cloud IDE

Running the lab

This lab is designed to be completed using a Cloud IDE environment. You will be writing and executing 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 Integrated Development Environment (IDE).

Important notice about this lab environment

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

Key terms

- Plugin: A package that provides access to device and platform-specific functionalities, such as camera access, GPS, and file storage.
- `pub.dev`: The official Dart package repository where plugins are hosted.
- `pubspec.yaml`: A configuration file for a Flutter project that specifies its dependencies.

Prerequisites

Before starting this lab, ensure that you have the following setup in your Cloud IDE:

1. Create a new Flutter project:

```
flutter create flutter_plugin_lab
cd flutter_plugin_lab
```

Step 1: Add the url_launcher plugin to your project

Before diving into the code, here are a few points to understand what you are about to do:

- **Create a Flutter app** that uses the `url_launcher` plugin to open a web URL.
- **Add an `ElevatedButton`** to the app's main screen that, when tapped, opens the Flutter website.
- **Use Dart's asynchronous features** to handle the process of launching the URL and managing potential errors.

1. Open your `pubspec.yaml` file and add the `url_launcher` plugin under the dependencies section:

```
dependencies:
  flutter:
```

```
    sdk: flutter
  url_launcher: ^6.0.3
```

- `url_launcher` is a plugin that allows you to open URLs in a browser or another application.

2. Save the `pubspec.yaml` file and install the plugin by running

```
flutter pub get
```

- `flutter pub get` fetches the dependencies specified in `pubspec.yaml`.

3. Open the `lib/main.dart` file in your project and import the `url_launcher` plugin:

```
import 'package:flutter/material.dart';
import 'package:url_launcher/url_launcher.dart';
```

- The `import` statement includes the `url_launcher` plugin in your project.

4. Create a function to launch a URL.

```
import 'package:flutter/material.dart';
import 'package:url_launcher/url_launcher.dart';
void main() {
  runApp(const MyApp());
}
class MyApp extends StatelessWidget {
  const MyApp({super.key});
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Plugin Lab',
      home: Scaffold(
        appBar: AppBar(
          title: const Text('Explore Plugins in Flutter'),
        ),
        body: Center(
          child: ElevatedButton(
            onPressed: _launchURL,
            child: const Text('Open Flutter Website'),
          ),
        ),
      ),
    );
  }
}
void _launchURL() async {
  const url = 'https://flutter.dev';
  if (await canLaunch(url)) {
    await launch(url);
  } else {
    throw 'Could not launch $url';
  }
}
```

Here's an explanation of the key parts of the code:

- `import 'package:flutter/material.dart';`: Imports the Flutter material package, which provides the necessary widgets and tools to build the app.
- `import 'package:url_launcher/url_launcher.dart';`: Imports the `url_launcher` package, which is required to open URLs in a browser or another app.
- `void main() { runApp(const MyApp()); }`: The `main` function is the entry point of the app. It calls the `runApp` method, which inflates the given widget (`MyApp`) and attaches it to the screen.
- `class MyApp extends StatelessWidget`: Defines a new stateless widget called `MyApp`. A stateless widget is a widget that does not require mutable state.
- `MaterialApp` **widget**: This widget is the root of the application. It provides essential settings such as `title` and `home`, which represents the main page of the app.
- `Scaffold` **widget**: This widget provides a high-level structure for the app, including the `AppBar` at the top and the main content area (`body`). This is where the button and other UI components are placed.
- `AppBar` **widget**: This widget creates a top bar with the title 'Explore Plugins in Flutter'.
- `Center` **widget**: This widget centers its child widget (the button) within the body of the screen.
- `ElevatedButton` **widget**: A button that, when pressed, triggers the `_launchURL` function.
  - `onPressed: _launchURL`: Defines what happens when the button is tapped. In this case, it calls the `_launchURL` function.
  - `child: const Text('Open Flutter Website')`: Sets the text displayed on the button.
- `_launchURL` **function**: An asynchronous function that handles the URL launching logic:
  - `const url = 'https://flutter.dev';`: Defines the URL to be opened.
  - `if (await canLaunch(url)) { ... }`: Checks if the URL can be launched using the `canLaunch` function provided by the `url_launcher` package.
  - `await launch(url);`: If the URL can be launched, this line launches the URL in the default browser.
  - `else { throw 'Could not launch $url'; }`: If the URL cannot be launched, an exception is thrown with an appropriate message.

Step 2: Run the app

Flutter provides `hot reload` that makes it very easy to test the code as you are developing it. That funcationality however does not work in the Cloud IDE. We have developed a `hot_reload.sh` script that essentially does the same thing. Please follow these setps to run the flutter application:

1. Change to the home directory

   ```
   cd /home/project
   ```

2. Set the `PROJECT_DIR` variable. This should be set to the `lib` directory of your flutter app folder.

   ```
   export PROJECT_DIR=/home/project/flutter_plugin_lab/lib
   ```

3. Get the script and save to the `/home/project` folder.

   ```
   wget -O /home/project/hot_reload.sh https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/S3cw83zxbxBhfixSSMK1Uw/hot-reload.sh
   ```

4. Make the script executable by using the `chmod` command.

   ```
   chmod +x ./hot_reload.sh
   ```

5. Run the command.

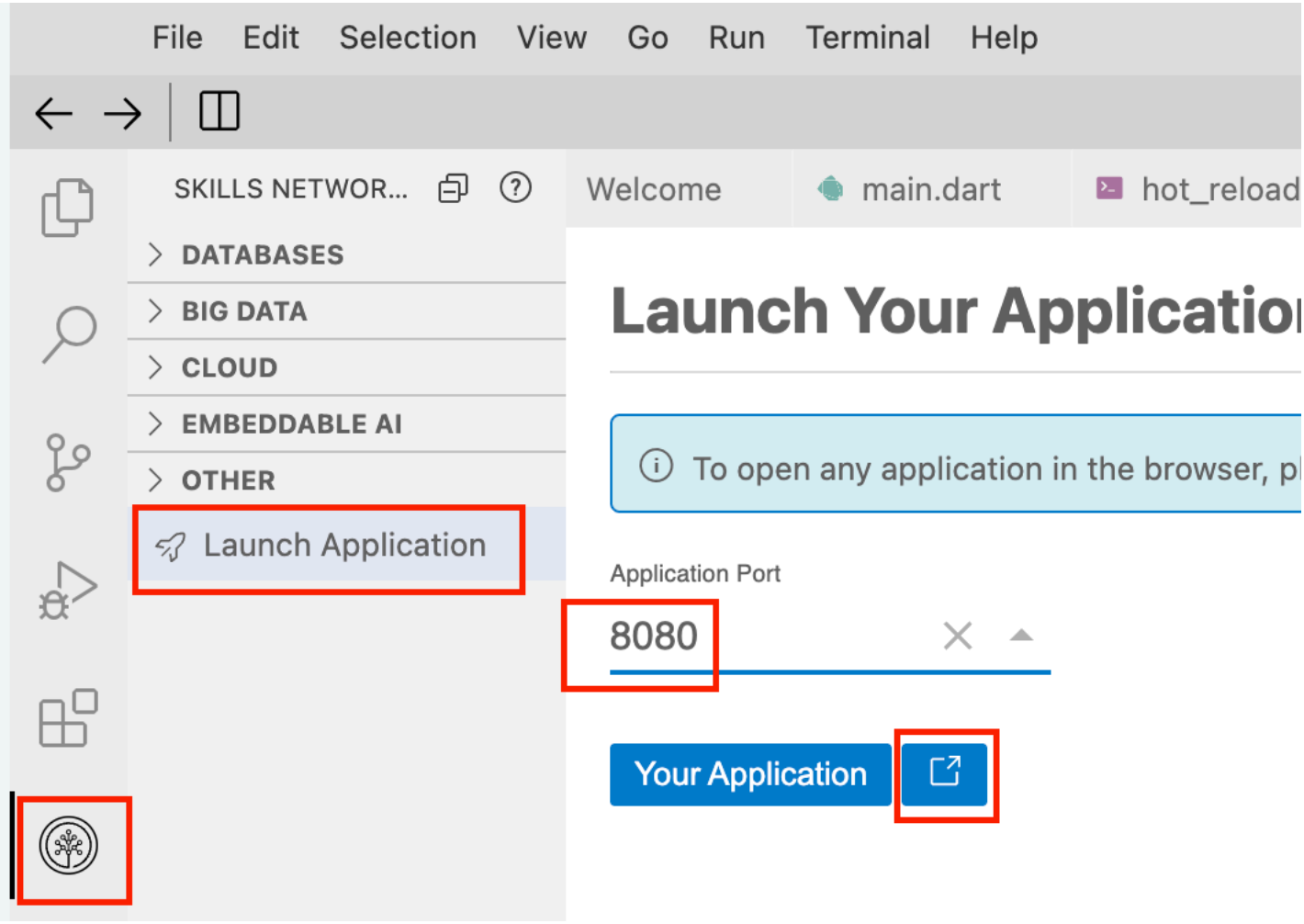
```
./hot_reload.sh
```

You should see output like this:

```
theia@theia-captainfedo1:/home/project$ ./hot_reload.sh
Using PROJECT_DIR: /home/project/flutter_plugin_lab/lib
inotify-tools is already installed.
lsof is already installed.
Starting Flutter web server...
Port 8080 is free.
Compiling lib/main.dart for the Web...                             984ms
✓ Built build/web
Flutter server started with PID 17675
Setting up watches.  Beware: since -r was given, this may take a while!
Watches established.
Serving HTTP on 0.0.0.0 port 8080 (http://0.0.0.0:8080/) ...
```

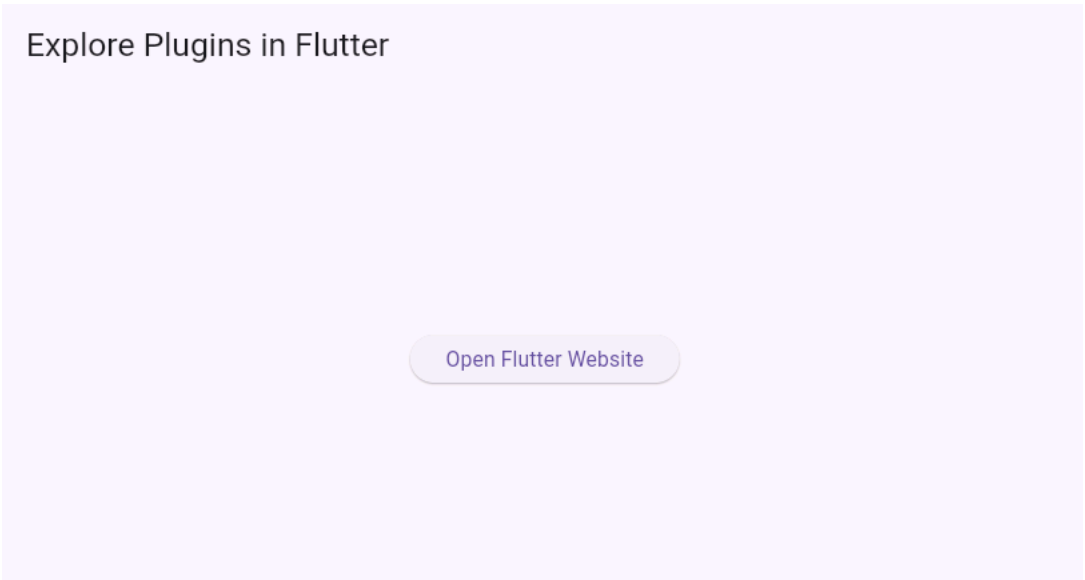
6. Launch the application

- open the `Skills Network Toolbox` in the let bar of the Cloud IDE.
- click on `Launch Application` button
- enter `8080` as the port
- select the `launch in new window` to open the applicaiton in a new tab



7. Test your app:

   You should see a simple page with `Open Flutter Website` button. Clicking this button will open the Flutter website in a new tab.



Step 3: Use the shared_preferences plugin to store data locally

In this code, you will learn to:

- Create a Flutter app that uses the `shared_preferences` plugin to store and retrieve data locally on the device.
- Implement a simple user interface with a text field to enter data and two buttons to save and load the data.
- Handle asynchronous operations to read from and write to the local storage.

1. Open your `pubspec.yaml` file and add the `shared_preferences` plugin under the dependencies section:

```
dependencies:
  flutter:
    sdk: flutter
  shared_preferences: ^2.0.6
```

- `shared_preferences` is a plugin that provides a way to persist simple data locally.

2. Save the `pubspec.yaml` file and install the plugin by running:

```
flutter clean
flutter pub get
```

3. Import the `shared_preferences` plugin into your Dart code in `lib/main.dart` file:

```
import 'package:flutter/material.dart';
import 'package:shared_preferences/shared_preferences.dart';
```

4. Update the `MyApp` widget to include a form that saves and retrieves user data:

```
import 'package:flutter/material.dart';
import 'package:shared_preferences/shared_preferences.dart';
void main() {
  runApp(const MyApp());
}
class MyApp extends StatefulWidget {
  const MyApp({super.key});
  @override
  _MyAppState createState() => _MyAppState();
}
class _MyAppState extends State<MyApp> {
  TextEditingController _controller = TextEditingController();
  String _storedValue = '';
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Plugin Lab',
      home: Scaffold(
        appBar: AppBar(
          title: const Text('Explore Plugins in Flutter'),
        ),
        body: Padding(
          padding: const EdgeInsets.all(16.0),
          child: Column(
            children: [
           TextField(
             controller: _controller,
             decoration: const InputDecoration(labelText: 'Enter some text'),
           ),
           const SizedBox(height: 16), // Adds space between TextField and Save button
           ElevatedButton(
             onPressed: _saveData,
             child: const Text('Save Data'),
           ),
           const SizedBox(height: 16), // Adds space between Save and Load button
           ElevatedButton(
             onPressed: _loadData,
             child: const Text('Load Data'),
           ),
           const SizedBox(height: 16), // Adds space between Load button and Text
           Text('Stored Value: $_storedValue'),
         ],
          ),
        ),
      ),
```

```
      );
    }
    void _saveData() async {
      SharedPreferences prefs = await SharedPreferences.getInstance();
      await prefs.setString('myKey', _controller.text);
    }
    void _loadData() async {
      SharedPreferences prefs = await SharedPreferences.getInstance();
      setState(() {
        _storedValue = prefs.getString('myKey') ?? 'No Data';
      });
    }
  }
```

Here's an explanation of the key parts of the code:

- `import 'package:flutter/material.dart';`: Imports the Flutter material package, which provides the widgets and tools needed to build the user interface.
- `import 'package:shared_preferences/shared_preferences.dart';`: Imports the `shared_preferences` package, which allows you to store data locally on the device.
- `void main() { runApp(const MyApp()); }`: The `main` function is the entry point of the app. It calls the `runApp` method, which inflates the given widget (`MyApp`) and attaches it to the screen.
- `class MyApp extends StatefulWidget`: Defines a new stateful widget called `MyApp`. A stateful widget is a widget that can maintain a mutable state over time.
- `_MyAppState` class: Manages the state of the `MyApp` widget, holding the data and logic to update the UI.
- `TextEditingController _controller`: A controller used to read the current value of the text field and manage the text input.
- `String _storedValue`: A variable to store the value retrieved from shared preferences.
- `MaterialApp` Widget: This widget is the root of the application. It provides essential settings such as `title` and `home`, which represents the main page of the app.
- `Scaffold` Widget: Provides a high-level structure for the app, including the `AppBar` at the top and the main content area (`body`).
- `Padding` Widget: Adds padding around the child widget (`Column`) to provide some space from the edges of the screen.
- `Column` Widget: Arranges its children (the text field, buttons, and text) in a vertical layout.
- `TextField` Widget: Allows the user to enter text.
  - `controller: _controller`: Connects the `TextField` to the `_controller` to manage the input.
  - `decoration`: Adds a label to the text field.
- `SizedBox(height: 16)`: Adds fixed vertical space between widgets (such as between the text field and buttons).
- `ElevatedButton` Widgets: Two buttons, “Save Data” and “Load Data”, that trigger functions when tapped.
  - `onPressed: _saveData`: Calls the `_saveData` function when the “Save Data” button is tapped.
  - `onPressed: _loadData`: Calls the `_loadData` function when the “Load Data” button is tapped.
- `Text` Widget: Displays the stored value retrieved from shared preferences.
- `_saveData` Function: An asynchronous function that saves the data entered in the text field to local storage using shared preferences.
  - `SharedPreferences prefs = await SharedPreferences.getInstance();`: Retrieves an instance of the shared preferences.
  - `await prefs.setString('myKey', _controller.text);`: Saves the text entered in the text field under the key `'myKey'`.
- `_loadData` Function: An asynchronous function that retrieves the stored data from local storage using shared preferences.
  - `SharedPreferences prefs = await SharedPreferences.getInstance();`: Retrieves an instance of the shared preferences.
  - `_storedValue = prefs.getString('myKey') ?? 'No Data';`: Loads the stored text under the key `'myKey'` or sets `'No Data'` if no data is found.

Step 4: Run the app

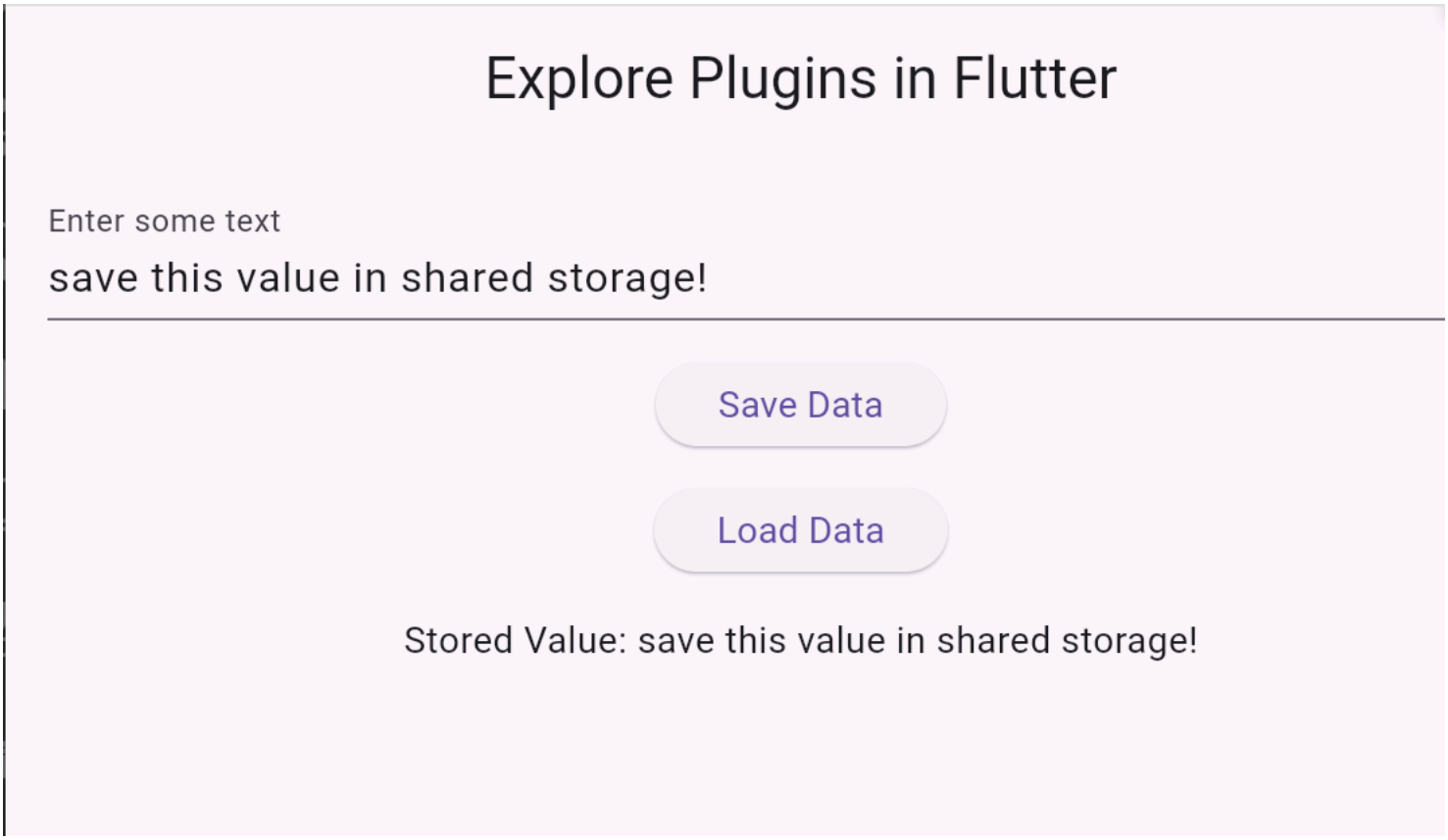
If you already have the app running from before, you don’t have to do anything. If you quit that process for some reason, run the script again from the `/home/project` folder.

```
cd /home/project && ./hot_reload.sh
```

Once the app starts, you can refresh the page if the tab is already open, or follow the instructions to launch application on port `8080` as before.

Test your app

You should see a simple page with a text box to enter your input. The `save` button will save the input to shared preferences and the `load` button will show the stored value in the label. To test, simply fill out the input, select `store` and then `load`.



Conclusion and next steps

Congratulations on completing this lab! You have learned how to find, add, and use plugins in Flutter, including creating your own custom plugin to interact with native code.

Now that you've completed the lab on exploring plugins in Flutter, here are a few next steps that are directly related to the contents of this lab:

1. Experiment with Other Plugins
   - Explore other common Flutter plugins such as path_provider for accessing the file system or image_picker for selecting images from the device's gallery. These plugins will allow you to extend your app's functionality further.
   - Example: Add the path_provider plugin to store a file on the device and retrieve it later.
2. Handle Errors Gracefully
   - While you've already implemented error handling with the url_launcher plugin (i.e., throwing an exception if the URL fails to open), try improving this further by showing user-friendly error messages via Flutter's SnackBar widget or a dialog box.
   - Example: When the URL fails to launch or data retrieval fails, display a SnackBar to notify the user.

Author(s)

Skills Network