```
class InputForm extends StatefulWidget {
  const InputForm({super.key});

  @override
  State<InputForm> createState() => _InputFormState();
}

class _InputFormState extends State<InputForm> {

  final _formKey = GlobalKey<FormState>();
  /* ◆ TextEditingController
  TextField Widgetの入力文字や選択文字を取得、変更する機能を持つ */
  final _textEditingController = TextEditingController(); ——❶

  @override
  Widget build(BuildContext context) {
    return Form(
      key: _formKey,
      child: Column(
        mainAxisAlignment: MainAxisAlignment.center,
        children: [
          Padding(
            padding: const EdgeInsets.symmetric(horizontal: 16),
            child: TextFormField(
              controller: _textEditingController, ——❷
              maxLines: 5,
              decoration: const InputDecoration(
                hintText: '文章を入力してください',
              ),
              validator: (value) {
                if (value == null || value.isEmpty) {
                  return '文章が入力されていません';
                }
                return null;
              },
            ),
          ),
          const SizedBox(height: 20),
          ElevatedButton(
            onPressed: () {
              final formState = _formKey.currentState!;
              if (!formState.validate()) {
                return;
              }
              debugPrint('text = ${_textEditingController.text}'); ——❸
            },
```

```
            child: const Text(
              '変換',
            ),
          ),
        ],
      ),
    );
  }

  @override
  void dispose() {
    _textEditingController.dispose(); ——❹
    super.dispose();
  }
}
```

クラスメンバにTextEditingControllerを加え(❶)、TextFormFieldウィジェットにパラメータとして渡しました(❷)。これでTextEditingControllerからTextFormFieldウィジェットの入力文字が取得できます。

バリデーションを通過した後、入力文字をログに出力するコードを追加しました(❸)。

TextEditingControllerクラスは不要になったら忘れずにdisposeメソッドを呼び出します(❹)。これにより、メモリリークのリスクを回避します。

_InputFormStateクラスのdisposeメソッドはStateのライフサイクルメソッドの一つで、StatefulWidgetが破棄されるときに呼び出されます。InputFormウィジェットはsetStateを呼び出して自身の状態を更新することはありませんが、disposeメソッドをオーバーライドしてTextEditingControllerクラスを破棄するためにStatefulWidgetを継承しました。

8.6 ひらがな化するWeb APIを呼び出す実装をする

入力文字のひらがな変換にはgooラボのひらがな化API[注1]を利用させてもらいます。APIの利用には利用登録とアプリケーションIDの取得が必要になり

注1 https://labs.goo.ne.jp/api/jp/hiragana-translation/

ます。詳しくは公式Webサイト[注2]をご覧ください。

リクエスト、レスポンスオブジェクトを定義する

APIのリクエストパラメータはJSON形式で送信します。json_serializableパッケージを利用して、JSONを型安全に扱いやすくするためのデータ型を定義します。libフォルダの配下にdata.dartという新しいファイルを追加し、以下のコードを記述します。

```
./lib/data.dart
import 'package:json_annotation/json_annotation.dart'; ―❶

part 'data.g.dart'; ―❷

@JsonSerializable(fieldRename: FieldRename.snake) ―❸
class Request { ―❹
  const Request({
    required this.appId,
    required this.sentence,
    this.outputType = 'hiragana', ―❺
  });

  final String appId;
  final String sentence;
  final String outputType;

  Map<String, Object?> toJson() => _$RequestToJson(this); ―❻
}
```

Requestクラスを定義しました(❹)。@JsonSerializableアノテーションを付与することで、json_serializableパッケージがJSONのシリアライズ、デシリアライズのコードを生成します(❸)。❶ではアノテーションを参照するため、json_annotation.dartをインポートしています。

Requestクラスのフィールドは、appId、sentence、outputTypeと3つ定義し、Dartの慣習にのっとってキャメルケースで命名しました。しかし、APIのリクエストパラメータはスネークケースです。@JsonSerializableアノテーションのfieldRenameプロパティにFieldRename.snakeを指定することで、JSONをシリアライズ、デシリアライズする際に、フィールド名をスネークケースに変換するよう指定しています。

注2 https://labs.goo.ne.jp/apiusage/

今回のアプリではoutputTypeは固定値なので、コンストラクタのデフォルト値を設定しました(❺)。

Requestクラスを Map形式に変換するためのtoJsonメソッドを定義しました(❻)。メソッドの本体はjson_serializableパッケージが生成し、_$ + クラス名 + ToJsonという命名規則になります。このメソッドを参照するため、part命令文でdata.g.dartをインポートしています(❷)。

実装が完了したらコード生成のために、以下のコマンドを実行してください。

```
$ flutter packages pub run build_runner build
```

同じように、レスポンスオブジェクトも定義します。data.dartに以下のコードを追加します。

./lib/data.dart

```
// 省略

@JsonSerializable(fieldRename: FieldRename.snake)
class Response {

  const Response({
    required this.converted,
  });

  final String converted;

  factory Response.fromJson(Map<String, Object?> json) => _$ResponseFromJson(jso
n); ——❶
}
```

Responseクラスを定義しました。convertedフィールドは変換後のひらがな文字列が入ります。Responseクラスのインスタンスを JSONから生成するための factoryコンストラクタを定義しました(❶)。こちらも json_serializableパッケージが生成します。

実装が完了したらのコード生成のために、以下のコマンドを実行しておきましょう。

```
$ flutter packages pub run build_runner build
```

アプリケーションIDを設定する

APIのリクエストにはアプリケーションIDが必要です。gooラボのひらがな化API[注3]のページから利用登録を行い、アプリケーションIDを取得してください。今回はアプリケーションIDをハードコーディングせずに、環境変数を利用する方法として第4章で解説したdart-define-from-fileのしくみで扱うことにします。

define/env.jsonというJSONファイルを作成し、以下のようにアプリケーションIDを記述します(YOUR_APP_IDの代わりに取得したIDを入れる)。

./define/env.json
```
{
  "appId": "YOUR_APP_ID"
}
```

そして、アプリの実行引数に--dart-define-from-file=define/env.jsonを指定します。詳しい方法は第4章をご覧ください。

なお、今回は設定とコードを分離する設計の観点で、dart-define-from-fileのしくみを利用しています。セキュリティの観点では、認証キーをdart-define-from-fileで扱うことがベストプラクティスとは言えません。アプリのセキュリティについてはリバースエンジニアリング、ルート化、中間者攻撃による通信の改ざんなど、さまざまな脅威があります。どの程度コストをかけてセキュアに扱うかは要件しだいと筆者は考えます。

本書ではこれ以上の解説は割愛しますが、少なくとも「認証キーはdart-define-from-fileで渡すのがベストプラクティス」という誤解を招かないように……という思いでここで補足しておきます。

Web APIを呼び出す

InputFormウィジェットの「変換」ボタンをタップしたときにWeb APIを呼び出すように実装します。

./lib/input_form.dart
```
import 'dart:convert'; ——❶

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

注3 https://labs.goo.ne.jp/api/jp/hiragana-translation/

```
import 'package:hiragana_converter/data.dart'; ——❷
import 'package:http/http.dart' as http; ——❸

// 省略

          ElevatedButton(
            onPressed: () async { ——❹
              final formState = _formKey.currentState!;
              if (!formState.validate()) {
                return;
              }
              final url = Uri.parse('https://labs.goo.ne.jp/api/hiragana');
              final headers = {'Content-Type': 'application/json'};          ——❺
              final request = Request(
                appId: const String.fromEnvironment('appId'),
                sentence: _textEditingController.text,                       ——❻
              );

              final result = await http.post(
                url,
                headers: headers,
                body: jsonEncode(request.toJson()),                          ——❼
              );

              final response = Response.fromJson(
                jsonDecode(result.body) as Map<String, Object?>,             ——❽
              );
              debugPrint('変換結果: ${response.converted}');
            },
            child: const Text(
              '変換',
            ),
          ),

// 省略
```

InputFormウィジェットの「変換」ボタンをタップしたときに呼び出されるコールバックで、Web APIを呼び出すコードを追加しました。最初にHTTPリクエストのURLやリクエストヘッダを生成します(❺)。次に先ほど定義したリクエストオブジェクトを生成します(❻)。Requestクラスを参照するため、data.dartをインポートしています(❷)。appIdはString.fromEnvironmentを使って環境変数から取得しています。実行引数が設定されていれば、環境変数にはdefine/env.jsonの内容が反映されます。

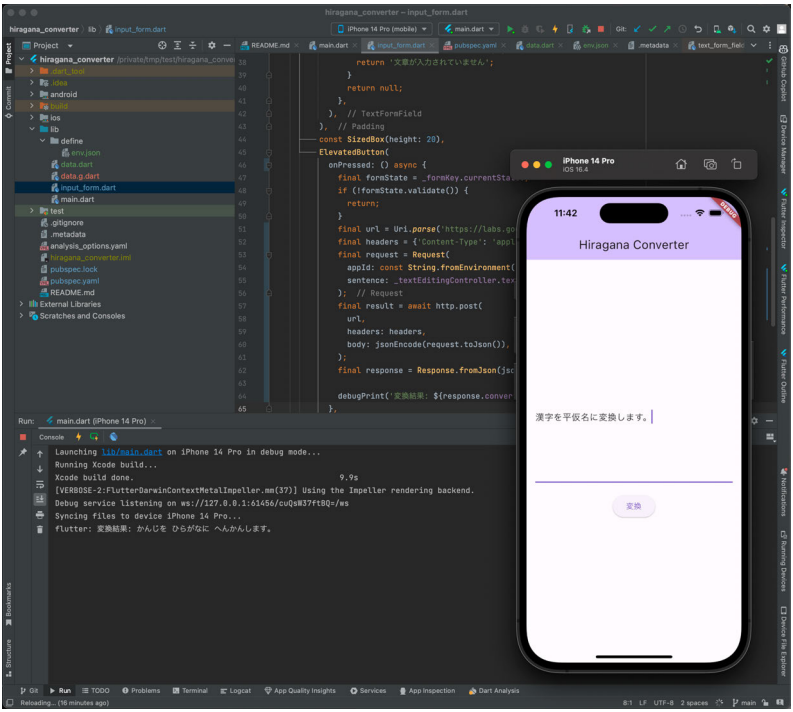
続いて、httpパッケージのpostメソッドを呼び出してWeb APIを呼び出し

ます(❼)。今回はhttpパッケージをインポートする際にasキーワードでhttpという別名を付けています(❸)。requestオブジェクトはtoJsonメソッドでMapに変換し、そこからさらにjsonEncode関数でJSON文字列に変換しています。jsonEncode関数を参照するため、組み込みパッケージのdart:convertをインポートしています(❶)。postメソッドの戻り値はFutureなので、awaitキーワードを付けて非同期処理の完了を待ち、onPressedコールバックにasyncを付与しています(❹)。

最後に、APIのレスポンスをデシリアライズして、変換結果をログに出力しています(❽)。JSON文字列をjsonDecode関数でMapに変換し、そこからResponseオブジェクトを生成しています。

これでアプリを実行して、入力文字の変換結果をログに出力できるようになりました。Android Studioであれば、「View」➡「Tool Windows」➡「Run」を選択し、ログを確認できます(**図8.7**)。

図8.7　**変換結果をログに出力した様子**



8.7 アプリの状態を管理する

Web APIのレスポンスを受け取り結果を表示したり、レスポンスを待つ間にインジケータを表示したりと、アプリの表示切り替えのために状態を管理します。

状態を表現するクラスを作成する

まずはアプリの状態をsealed classで表現してみましょう。libフォルダの配下にapp_state.dartという新しいファイルを追加し、以下のコードを記述します。

./lib/app_state.dart

```
sealed class AppState {
  const AppState();
}

class Input extends AppState {
  const Input(): super();
}

class Loading extends AppState {
  const Loading(): super();
}

class Data extends AppState {
  const Data(this.sentence);

  final String sentence;
}
```

❶ ❷ ❸ ❹

アプリの状態を表現する、AppStateというsealed classを定義しました(❶)。AppStateを継承したInput、Loading、Dataという3つのクラスを定義しました(❷、❸、❹)。Inputは入力状態、LoadingはWeb APIのレスポンス待ちの状態、DataはWeb APIのレスポンスを受け取った状態を表現します。

続いて、libフォルダの配下にapp_notifier_provider.dartという新しいファイルを追加し、以下のコードを記述します。