26. HTTP, APIs, REST & JSON

Introduction

Most Flutter projects involve HTTP communication between your app and some API on some server. Most of the time these server APIs are built to the REST design guidelines and the data will be transferred in the JSON format.

The purpose of this chapter is to learn about HTTP, APIs, REST and JSON before we hit the keyboard.

Asynchronous Communication

When your app communicates to and from a remote server using HTTP, it is doing so asynchronously. The app does not suddenly stop completely after it sends a request to the server. As mentioned in the chapter ‘More Advanced Dart’, the Dart language fully supports asynchronous programming, including Futures. The Flutter HTTP package (which we will cover soon) uses Futures to enable developers to communicate through HTTP asynchronously. Every time we communicate with the server using HTTP we don’t stop doing things in the app but we process the success or error response when it comes back to us.

HTTP

Introduction

The Hypertext Transfer Protocol (HTTP) is designed to enable communications between clients and servers. HTTP works as a request-response protocol between a client and server.

A protocol describes how machines communicate with each other using messages. A protocol defines the format of these messages.

Tools

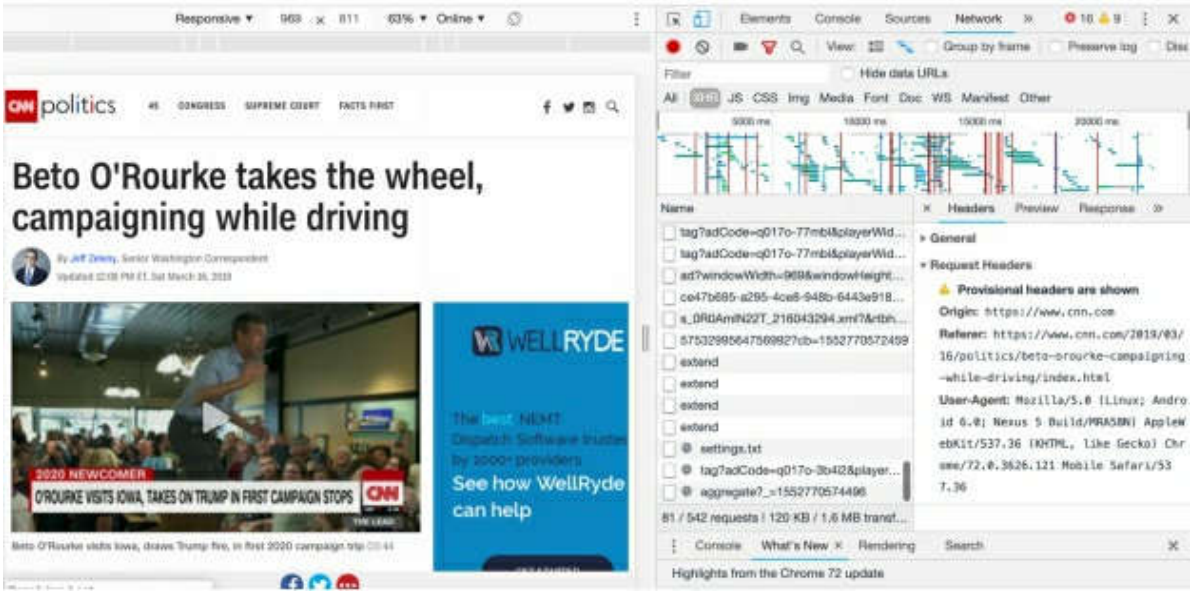
Introduction

One you are adept with Flutter you will end up spending considerable time writing code that communicates with servers using the HTTP. You may want to investigate these tools in advance, they will make your life easier.

Web Browser

You obviously already have one of these. If you want to see the HTTP protocol at work, open your browser, go to a website then use the hamburger menu to access the developer tools. Select the ‘network’ option to see

the network traffic inspector. In the image below you can see the network traffic inspector on the right side, with one request selected and viewed in more detail.



Postman

This tool will let you test HTTP requests to a server before you code them in Flutter. You can view the raw data and see what's going on.
Https://www.getpostman.com/

JSON Formatter

JSON is the data format you will be working with. You may also want to find a good online JSON formatter to make the JSON more readable.
Https://jsonformatter.curiousconcept.com/

Request

This is what your app will send to the server.

Response

This is what your app will receive back from the server.

Methods

HTTP methods have been around for a long time. The most-commonly-used HTTP methods are POST, GET, PUT, PATCH, and DELETE. GET is used most of all because you tend to access data more often than you change it.

GET request ‘method’. The ‘method’ describes what the app wants the server to do, what is the intent of the request. The most commonly used methods are ‘get’ and ‘post’. The ‘get’ method is used to request data from the server. The ‘post’ method is used to send data to the server, to save it or update it. The ‘put’ method is used to update data on the server. The ‘delete’ method is used to delete data on the server.

URI

This is the address of where the request is going to. A specific path on a specific server.
Example:
https://www.cnn.com/2019/03/16/politics/beto-orourke-campaigning-while-driving/index.html

Query Parameters

HTTP allows you to pass information to the server in the URL using query parameters.
Example:
http://localhost:4200/sockjs-node/info?t=1498649243238

Matrix Parameters

HTTP allows you to pass information to the server in the URL using matrix parameters. For Example:
http://localhost:4200/sockjs-node/info;t=1498649243238

Matrix parameters are similar to query strings but use a different pattern. They also act differently because (not having a '?') they can be cached.

Path Parameters

HTTP allows you to pass information to the server in

the URL using path parameters. For Example:
http://localhost:4200/api/badges/9243238

URI Encoding

Some characters cannot be part of a URL (for example spaces) and some other characters have a special meaning in a URL. To get around this, the URL syntax allows for encoding on parameters to ensure a valid URL.

Example:
The ‘space’ character between ‘Atlantic’ and ‘City’ is encoded to ‘%20’.
https://trailapi-trailapi.p.mashape.com/?q[city_cont]=Atlantic%20City

Status

This is part of the response. It indicates whether the request was successfully processed or not. Here are some of the HTTP status code values:

Code	Name	Description
1xx	Informational	
2xx	Success	
200	Ok	
3xx	Redirect	

	301	Moved permanently	
	302	Moved temporarily	
4xx		Request error	
	400	Bad request	The request could not be understood by the server.
	403	Forbidden	User not authorized to perform the requested operation.
	404	Not found	The requested resource could not be found at the given URI.
	405	Method not allowed	The request method is not allowed on the specified resource.
5xx		Server error	
	500	Internal server error	The server encountered an unexpected condition, preventing it to fulfill the request.
	503	Service unavailable	The server is temporarily unavailable, usually due to overloading or maintenance.

Header

HTTP headers allow the client and the server to pass additional information with the request or the response. A request header consists of key value pairs - a case-insensitive key followed by a colon ':', then by its value (without line breaks).

```
Host: localhost:3000
Connection: keep-alive
Content-type: application/json
```

Body

Introduction

The HTTP body allows the client and the server to pass additional information with the request or the response after the header.

Request

In the Request, HTTP bodies are not always required because a body of information is not always needed. GET and DELETE HTTP requests usually don’t need a body. POST, PUT and PATCH HTTP requests do - this is where the information to be created or modified is sent.

```
{ "name": "John", "age: 35 }
```

Response

The body is used to return information in the Response and it can get very large, with a considerable amount of data.

Example:

In this chapter's example HTTP code, we receive a response with a body containing the data for over 1000 employees.

APIs

When someone makes their API available to the world, they write the code for the api and they publish it to their HTTP web server. APIs are also known as web services.

Most APIs use the REST architectural style, which is a pattern of how you will communicate with the server over HTTP. APIs that conform to the REST architectural style mostly work in the same manner, with similar web addresses (URIs) and HTTP methods.

These similarities really help when going from one API to another.

REST

REST stands for Representational State Transfer.

REST gives us high level design guidelines and leave you to think of your own implementation.

REST APIs should be stateless.

In the past, web applications used to store session data for the user. For example, the user would login and this would start a session and information could be kept in this session until the user logged out. This session data could include who the user is, what access they have and any other required information.

Now, with more modern APIs and REST, access to servers is controlled through tokens or api keys. Also, every API call is stateless - every single request from the client to server is self-contained and contains all of the data to identify who made the request and all of the request data itself to perform the operation. Such a request cannot take advantage of any pre-existing session data on the server.

Determining the User - Who Made the Request to the API?

Tokens

In most apps with a login, when a user login occurs, he or she is returned a temporary token for access. This

token is encrypted and contains information about the user and the token itself (such as when it expires). This token can be refreshed every predetermined period of time (for example every 15 minutes). Whenever an API call is made from some device to the server, the token must be included in every single outgoing request header to the server. If the token is not present or invalid (they can expire) then the server returns an error code (usually a 401 or 403 HTTP code). If the token is good then the server knows that a valid logged-in user is using the app, the server has info about the user from the token and the API can perform its operation.

API Keys

If the user doesn't really need to login every time the app is used, an API key enables a registered user (for example a CAT API user) to be identified in the HTTP header as a valid user on every single outgoing request to the server. Like a token, this is validated and the server returns an error code if there is a problem with it.

No User Identification

Sometimes people publish APIs which don't need information about the user. For example, in this chapter we are going to use the dummy rest api here:

http://dummy.restapiexample.com/

How REST Uses URLs

In REST, the URL is used to determine what resource you are doing it to. For example: employees, orders etc.

Base URL

The base URL is the first part of the API, without the REST part. The REST part comes after the base URL. The base URL is usually the following:

- The domain. E.g. www.example.com.
- Optionally a suffix ‘api’ to indicate that the path is for API use only.
- Optionally a suffix for the name of the app the API was written for.
- Optionally it also has the API version.
- For example, for the dummy REST API it is http://dummy.restapiexample.com/api/v1

URL & Paths

The URL of the REST API can be composed of several parts, of paths. Think of it the URL as a path to the resource (the data).

- Example:

    - http://www.example.com/customers/33245
- Should be thought as:
    - Go to customer 33245.
    - Then go to order 8769 for that customer.
    - Then ‘go to line item 1’ for that order.

How REST Uses HTTP Method

In REST, the HTTP method is used to describe what you are doing. Getting data, posting new data (creating it), putting data (updating it), deleting it.

Accessing Data with a REST API

- URI
    - Identifies what data you are accessing.
    - A list of items.
        - This would be [base url] + the resource name. For example: http://www.example.com/products. This would usually return multiple projects.
        - The list of items could belong to another entity. Examples:
            - http://www.example.com/cus would return the list of orders for customer 33245.

            - http://www.example.com/cus would return the line items for order 123 for customer 33245.
    - A searched list of items.
        - The URL would be similar to the list of items above, plus some additional info on the end to specify the search.
        - Additional info.
            - You could add query strings or matrix / path parameters to the end of the url. For example: http://www.example.com/pro name=mark . This is the preferred way to do this but REST URLs are often open to interpretation.
            - You could add ‘/search’ then the search criteria to the end of the URL (or something similar). For example: http://www.example.com/pro would search for products by the name mark.

        - A single item.
            - The URL would be similar to the list of items plus a slash then an identifier to identify the item. For example: http://www.example.com/products/6 would return product 66432.
            - The single item could belong to another entity. For example: http://www.example.com/customers would return a single item, order 8769 for customer 33245.
- HTTP Method
    - You should use an HTTP ‘get’ method to access data through a REST API.
- HTTP Body
    - Not used.

Inserting Data with a REST API

- URI
    - Identifies what type of data you are inserting.
    - This would be the same as the URL to the list of items. Examples:
        - http://www.example.com/products

        - http://www.example.com/customers or http://www.example.com/orders (implementation is open to interpretation).
- HTTP Method
    - You should use an HTTP ‘put’ method to insert (or create) data through a REST Api.
- HTTP Body
    - You normally put the data required for the insert in the request body.

Updating Data with a REST API

- URI
    - Identifies what data you are updating.
    - This would be the same as the URL for accessing a single item. Examples:
        - http://www.example.com/products/6
        - http://www.example.com/customers

- HTTP Method
    - You use an HTTP ‘put’ method to update data through a REST Api.
- HTTP Body
    - You normally put the data required for the

update in the request body.

Deleting Data with a REST API

- URI
    - Identifies what data you are deleting.
    - This would be the same as the URL for accessing a single item. Examples:
        - http://www.example.com/products/6
        - http://www.example.com/customers
- HTTP Method
    - You should use an HTTP ‘delete’ method to delete data through a REST Api.
- HTTP Body
    - Not used.

JSON

JSON stands for JavaScript Object Notation. It is a data format used to pass data between the client and the server (in both directions). It is the same data format used by the JavaScript language. It uses a comma to separate items and a colon to separate the name of a property with the data for that property. It uses different types of brackets to denote objects and arrays.

JSON For Passing an Object

Containing Data.

The ‘{‘ and ‘}’ brackets are used to denote the start and end of an object.

```
{ "name":"John", "age":31, "city":"New York" }
```

JSON For Passing an Array

The ‘[‘ and ‘]’ brackets are used to denote the start and end of an array.

```
[ "Ford", "BMW", "Fiat"]
```

JSON For Passing an Array of Objects

The brackets are combined to create a cars object, which has two properties ‘Nissan’ and ‘Ford’. Each property has an array of models.

```
{
   "cars": {
      "Nissan": [
         {"model":"Sentra", "doors":4},
         {"model":"Maxima", "doors":4}
      ],
      "Ford": [
         {"model":"Taurus", "doors":4},
         {"model":"Escort", "doors":4}
      ]
   }
```

```
}
```

27. Flutter with HTTP, APIs, REST & JSON

Introduction

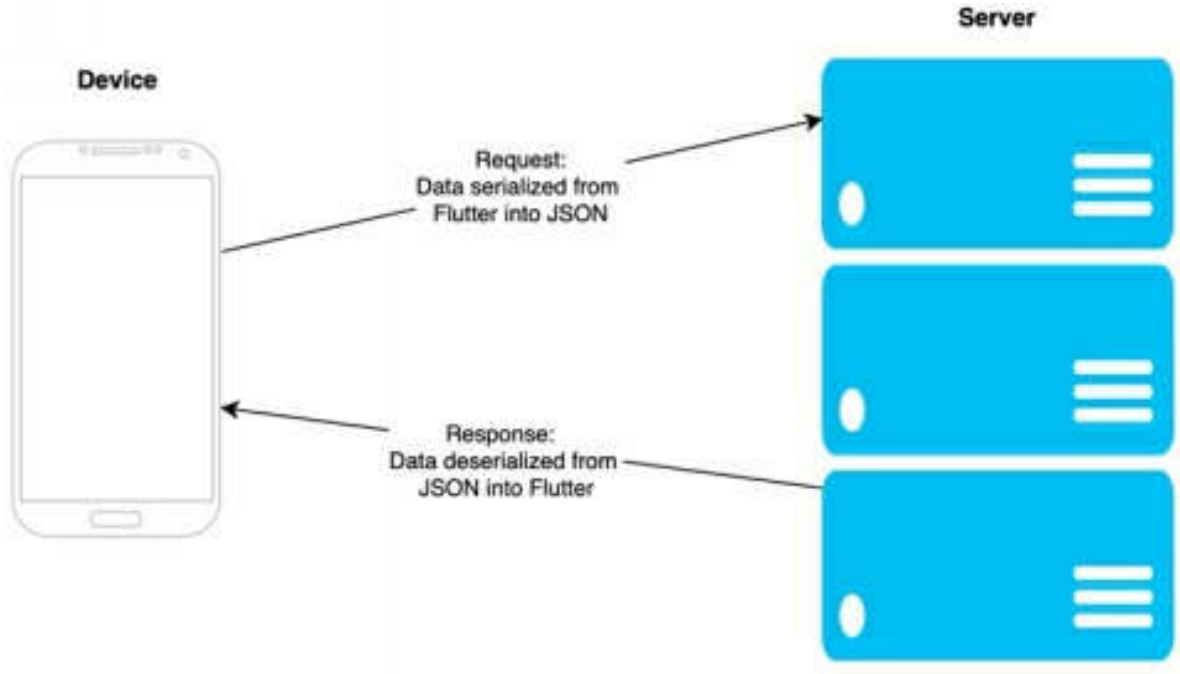
In the previous chapter we learnt about HTTP, APIs, REST and JSON.

The purpose of this chapter is to write Flutter code that communicates with APIs over HTTP using REST with JSON as the data format.

Flutter & JSON

Introduction

So, we know that we communicate with servers using the HTTP protocol, using JSON as the data format.



Request

When the app makes an outgoing request to an API on a server, it often needs to convert Flutter data (for example data in a form) into JSON. This conversion from Flutter data into JSON data is called serializing.

Response

When the server responds back, the app needs to convert JSON data into Flutter data. This conversion from JSON data back to Flutter data is called deserializing.

Serializing & Deserializing JSON.

So, we know we have to convert data between the JSON and Flutter.

- By JSON we mean a string of JSON.
- By Flutter we mean ‘data in a Dart class in our Flutter app’.

Two Ways of Serializing & Deserializing JSON

These are the two main ways of serializing & deserializing JSON in a Flutter App:

1. Generating code for Serializing & Deserializing
    - Pluses.
        - You don’t have to write the code.
        - Its generated code, it doesn’t make mistakes.
    - Minuses.
        - It’s not super-simple to setup, you need to know how it works.
        - It doesn’t work with complicated cases as well as coding them.
2. Manually writing code for Serializing & Deserializing
    - Pluses.
        - You have to write the code.
        - You can code the more complex Serialization & Deserialization scenarios.

    - Minuses.
        - There will be bugs.
        - It’s not super-simple to code, you need to know how it works.

Remember that You Can Combine the Two!

You can follow the 80 - 20 rule.
Do 80% the simple easiest way, generating the code for the serialization & deserialization of simple objects.
When you get to the more difficult 20% you can handcraft your own code to serialize and deserialize more complex objects.

The code examples follow this rule. We do the easy stuff using the code generator (simple serialization & deserialization) and the hard stuff (recursive serialization & deserialization) in the handwritten code.

Generating Code for Serializing & Deserializing

Introduction

This approach uses two packages:

- The ‘json_serializable’ package to generate the serialization & deserialization code for us.

- The ‘build_runner’ package to work with the ‘json_serializable’ package generate the code files.

Step 1 – Add Dependencies to Projects

Modify the project dependency file ‘pubspec.yaml’ to include two additional developer dependencies - build_runner and json_serializable:

```
dev_dependencies:
  flutter_test:
    sdk: flutter

  build_runner:
  json_serializable: ^0.5.0
```

Then you need to command Flutter to go get the dependencies:

```
flutter packages get
```

Step 2 – Amend the classes to be Serialized & Deserialized

Annotate the classes to be serialized & deserialized to include the import and annotations. In the example, this class is contained in the ‘main.dart’ file.

- Import the annotation.
- Add a @JsonSerializable() annotation just before

the class declaration.

- Add field annotations just before the field declarations.
    - These aren’t necessary if the JSON field name stays the same as the Dart field name.
    - The @JsonKey annotation declares the JSON name for the field if you want it to be different from the field name.

```
import 'package:json_annotation/json_annotation.dart';
```

…

```
@JsonSerializable()
class Person {
  final String name;
  @JsonKey(name: "addr1")
  final String addressLine1;
  @JsonKey(name: "city")
  final String addressCity;
  @JsonKey(name: "state")
  final String addressState;

  Person(this.name, this.addressLine1, this.addressCity,
this.addressState);

  @override
  String toString() {
```

```
    return 'Person{name: $name, addressLine1: $addressLine1,
addressCity: $addressCity, addressState: $addressState}';
  }
}
```

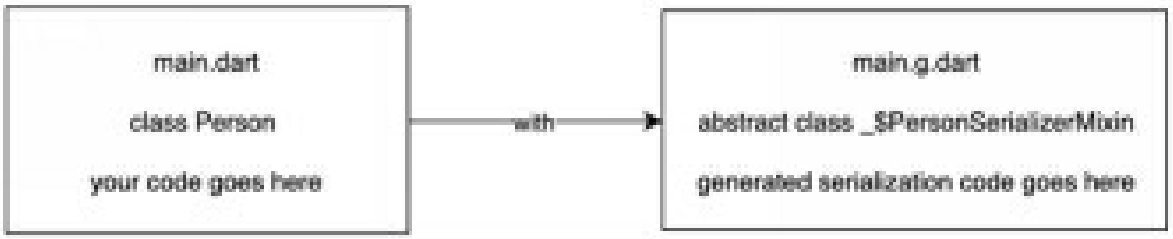
Step 3 – Generate the Serialization & Deserialization Code '.g.dart' Files

Run the following command line in the project root:

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

This should generate a '.g.dart' file in the project for each file that you modified in Step 2. Note that these files contain 'Mixins', Dart classes that contain code that can be incorporated into other classes without the use of inheritance. Please take a look at the Mixins chapter.

In the example, this generates a file 'main.g.dart' to match the 'main.dart' file:



Step 4 – Amend the classes to be Serialized & Deserialized

Now we need to go back to the classes that we modified in step 2 and we need to modify them to

utilize the generated code. We do this by first using a ‘part’ annotation to import the generated code. Then we use a mixin to combine the existing class and the generated class together.

- We insert a ‘part’ annotation for each file generated in Step 3. The ‘part’ annotation is used to inject content from another file. In the example file ‘main.dart’, we use this annotation to inject the content from the ‘main.g.dart’ file.

```
part 'main.g.dart';
```

- We modify the class declarations to extend the Object class with the Mixin (the abstract class) from the generated code (you may need to look in the ‘.g.dart’ files to get the mixin names). In the example, we change the class declaration to the following (changes in bold):

```
class Person extends Object with _$PersonSerializerMixin {
```

Done

That’s it, you should be done.
Make sure that you re-run the following command in your project root everytime you change something:

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

Example – ‘serialize_with_generated_code’

This app creates a Person object for a person and displays a ‘toString()’ of the object below in black. It also serializes that object and displays the JSON in underneath in red. There is a ‘Copy’ button to copy the JSON to the clipboard so you can paste it into an online JSON formatter.

Remember that this should won’t work recursively, unlike the example with the manually-written code.

Source Code

```
import 'dart:convert';

import 'package:flutter/material.dart';
import 'package:flutter/services.dart';
import 'package:json_annotation/json_annotation.dart';

part 'main.g.dart';

void main() => runApp(MyApp());

@JsonSerializable()
class Person extends Object with _$PersonSerializerMixin {
  final String name;
  @JsonKey(name: "addr1")
  final String addressLine1;
  @JsonKey(name: "city")
  final String addressCity;
  @JsonKey(name: "state")
  final String addressState;

  const Person(
      this.name, this.addressLine1, this.addressCity,
this.addressState);

  @override
  String toString() {
    return 'Person{name: $name, addressLine1: $addressLine1,
addressCity: $addressCity, addressState: $addressState}';
```

```
  }
}

class MyApp extends StatelessWidget {
  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Demo',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: HomeWidget(),
    );
  }
}

class HomeWidget extends StatelessWidget {
  static const Person _person =
      Person("John Brown", "9621 Roberts Avenue",
"Birmingham", "AL");

  @override
  Widget build(BuildContext context) {
    return Scaffold(
        appBar: AppBar(
          title: Text("Serialization"),
```

```
      ),
      body: Center(
        child: Padding(
          child: ListView(
            children: <Widget>[
              Padding(
                  child: Text("Grandfather:\n${_person}"),
                  padding: EdgeInsets.only(top: 0.0)),
              Padding(
                  child: Text("Json
Encoded:\n${json.encode(_person)}",
                      style: TextStyle(color: Colors.red)),
                  padding: EdgeInsets.only(top: 10.0)),
              FlatButton(
                  child: Text("Copy"),
                  onPressed: (() {
                    Clipboard.setData(
                        ClipboardData(text:
"${json.encode(_person)}"));
                  })),
            ],
          ),
          padding: EdgeInsets.all(10.0),
        ),
      ));
  }
}
```

Example – ‘deserialize_with_generated_code’

This app lets you enter the JSON for a person then hit the floating button to deserialize it.

- If successful, a ‘toString()’ of the Person object is displayed underneath (in black).
- If an error occurs (maybe you input bad JSON?), it is displayed underneath (in red).

Remember that this should won't work recursively, unlike the example with the manually-written code.

Source Code

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

part 'main.g.dart';
```

```
void main() => runApp(MyApp());

@JsonSerializable()
class Person extends Object with _$PersonSerializerMixin {
  final String name;
  @JsonKey(name: "addr1")
  final String addressLine1;
  @JsonKey(name: "city")
  final String addressCity;
  @JsonKey(name: "state")
  final String addressState;

  Person(this.name, this.addressLine1, this.addressCity,
this.addressState);

  @override
  String toString() {
    return 'Person{name: $name, addressLine1: $addressLine1,
addressCity: $addressCity, addressState: $addressState}';
  }
}

class MyApp extends StatelessWidget {
  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Demo',
```

```
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: HomeWidget(),
    );
  }
}

class HomeWidget extends StatefulWidget {
  HomeWidget({Key key}) : super(key: key);

  @override
  _HomeWidgetState createState() => _HomeWidgetState();
}

class _HomeWidgetState extends State<HomeWidget> {
  final _jsonTextController = TextEditingController();
  Person _person;
  String _error;

  _HomeWidgetState() {
    final String person =
        "{\"name\":\"Tracy Brown\", \"addr1\":\"9625 Roberts
Avenue\"," +
            "\"city\":\"Birmingham\", \"state\":\"AL\"}";

    _jsonTextController.text = person;
  }
```

```
  TextFormField _createJsonTextFormField() {
    return new TextFormField(
        validator: (value) {
          if (value.isEmpty) {
            return 'Please enter the json.';
          }
        },
        decoration: InputDecoration(
            border: OutlineInputBorder(),
            hintText: 'Json',
            labelText: 'Enter the json for a person.'),
        controller: _jsonTextController,
        autofocus: true,
        maxLines: 8,
        keyboardType: TextInputType.multiline);
  }

  _convertJsonToPerson() {
    _error = null;
    _person = null;
    setState(() {
      try {
        final String jsonText = _jsonTextController.text;
        debugPrint("JSON TEXT: ${jsonText}");
        var decoded = json.decode(jsonText); // text to map
        debugPrint("DECODED:  type: ${decoded.runtimeType}
value: ${decoded}");
```

```
      _person = _$PersonFromJson(decoded); // map to object
      debugPrint("PERSON OBJECT: type:
${_person.runtimeType} value: "
          "${_person}");
    } catch (e) {
      debugPrint("ERROR: ${e}");
      _error = e.toString();
    }
  });
 }

 @override
 Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: Text("Deserialization"),
    ),
    body: Center(
      child: Padding(
        child: ListView(
          children: <Widget>[
            _createJsonTextFormField(),
            Padding(
                padding: EdgeInsets.only(top: 10.0),
                child: Text(
                    _error == null ? '' : 'An error
occurred:\n\n${_error}',
```

```
                style: TextStyle(color: Colors.red))),
            Padding(
              padding: EdgeInsets.only(top: 10.0),
              child: Text(_person == null
                ? 'Person is null'
                : 'Converted to Person object:\n\n${_person}'))
          ],
        ),
        padding: EdgeInsets.all(10.0),
      ),
    ),
    floatingActionButton: FloatingActionButton(
      onPressed: _convertJsonToPerson,
      tooltip: 'Increment',
      child: Icon(Icons.refresh),
    ), // This trailing comma makes auto-formatting nicer for build methods.
  );
 }
}
```

Manually Writing Code for Serialization & Deserialization

Introduction

This approach uses the ‘json’ class in the core

‘dart.convert’ package to convert between maps and JSON strings.
When serializing an Object, we write code to convert the data in our class into a map so that the ‘json’ class can then convert it to a JSON string.
When deserializing an JSON string, we write code to convert the map into the data in our class.

Step 1 - Write Data Class Including ‘toJson’ & ‘fromJson’ Methods

- First of all, you need to write a Dart data class that will contain the data to be serialized and will contain the data after it has been deserialized.

- If serializing:
    - Write a ‘toJson’ method that returns a map from the data in that class (see ‘Person’ class for example).
- If deserializing:
    - Write a ‘fromJson’ factory method that creates an instance of the data class from a single map argument.

```
class Person {
  final String name;
  final String addressLine1;
```

```
final String addressCity;
final String addressState;
final List<Person> children;

const Person(this.name, this.addressLine1, this.addressCity,
    this.addressState, this.children);

Map<String, dynamic> toJson() {
  var map = {
    'name': name,
    'addr': addressLine1,
    'city': addressCity,
    'state': addressState,
    'children': children
  };
  return map;
}

factory Person.fromJson(Map<String, dynamic> json) {
  if (json == null) {
    throw FormatException("Null JSON.");
  }

  // Recursion. Convert children into list of Person objects.
  List<dynamic> decodedChildren = json['children'];
  List<Person> children = [];
  decodedChildren.forEach((decodedChild) {
    children.add(Person.fromJson(decodedChild));
```

```
    });

    return Person(
        json['name'], json['addr1'], json['city'], json['state'],
children);
  }

}
```

Step 2 - Add Code to Invoke Serialization / Deserialization of the Data Class

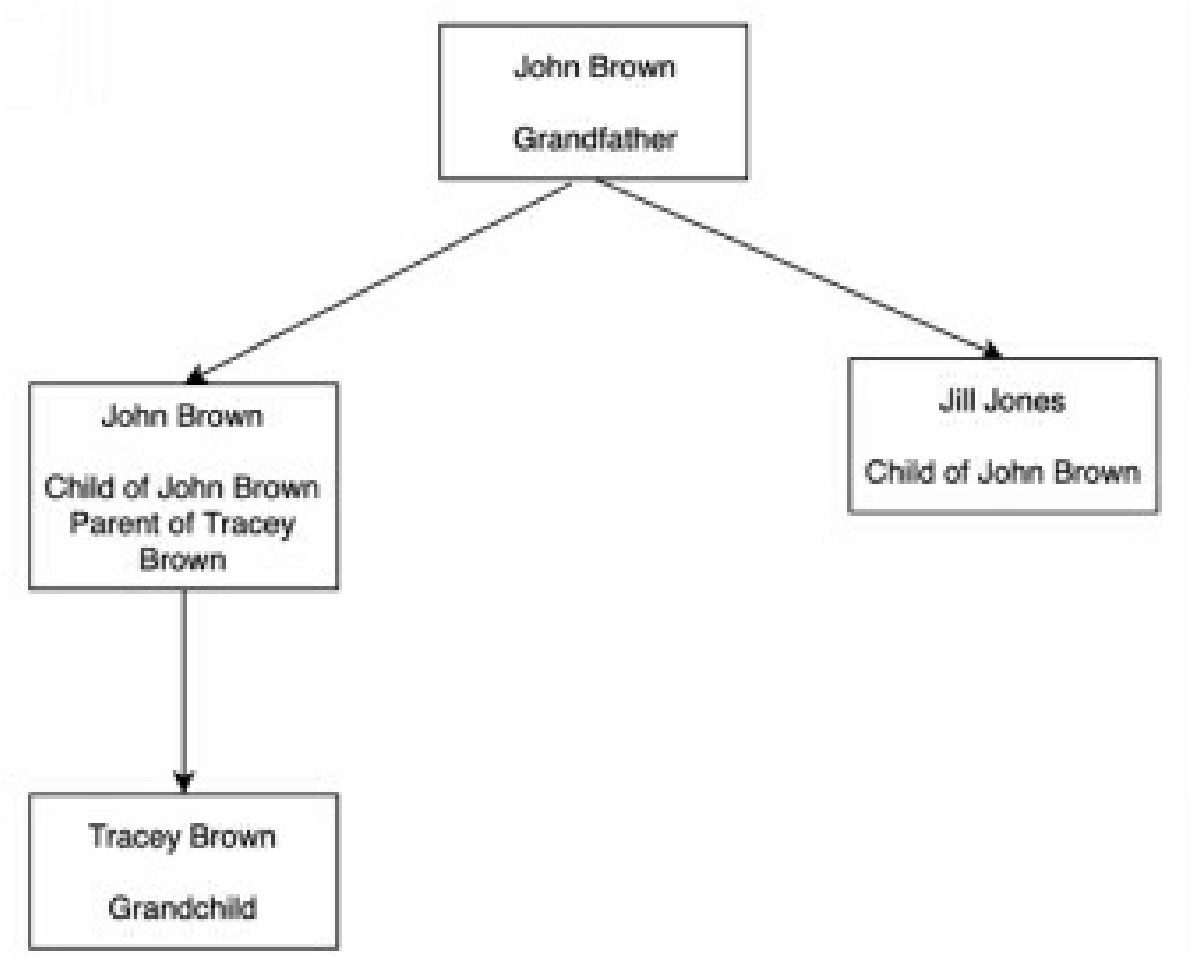
- If serializing:
    - Invoke ‘json.encode’ in the ‘json’ class in the core ‘dart.convert’ package.
        - The ‘json’ class invokes the ‘toJson’ method in your data class to create a map.
        - The ‘json’ class then converts the map to a JSON string.
- If deserializing:
    - Invoke ‘json.decode’ in the ‘json’ class in the core ‘dart.convert’ package to return a map.
        - The ‘json’ class will convert the JSON string into a map.
    - Invoke the factory ‘.fromJson’ method in

the data class to convert the map into an instance of the data class.

Examples – ‘serialize_manually’ & ‘deserialize_manually’

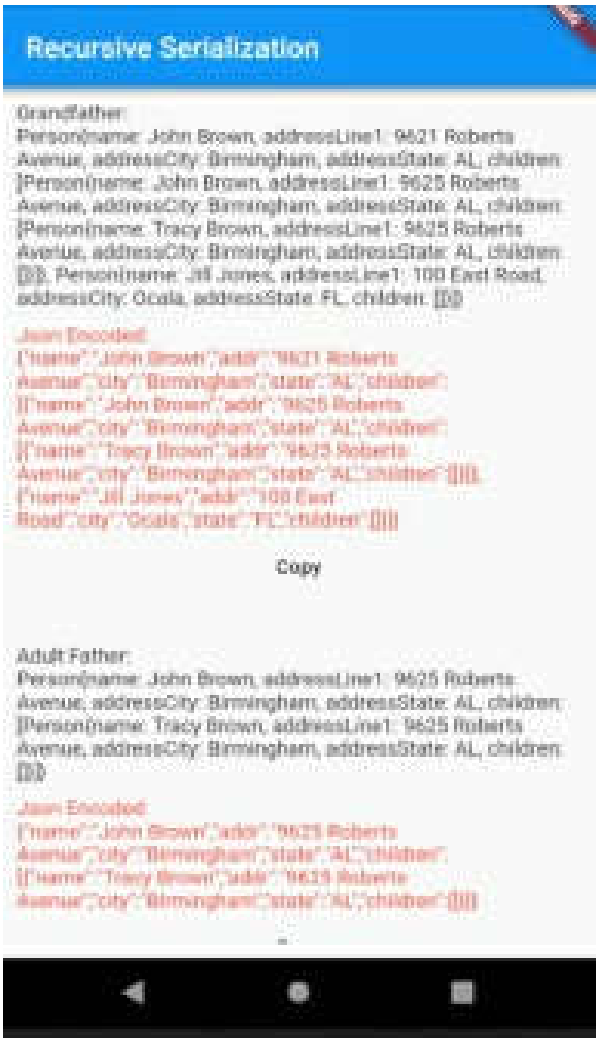
Both the examples below demonstrate something more complex: recursive manual serialization / deserialization. I tried to do this with the generated code but could not get it to work.

We demonstrate serializing & deserializing a Person object recursively. These Person objects can have children, which in turn can have children etc. In this example, we can have children and grandchildren.



Example – ‘serialize_manually’

This app creates Person objects for all the people in the family and displays a ‘toString()’ of each one (in black). It also deserializes each one, displaying the JSON in underneath (in red). There is a ‘Copy’ button to copy the JSON to the clipboard so you can paste it into an online JSON formatter.



Source Code

```
import 'dart:convert';

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

void main() => runApp(MyApp());

class Person {
  final String name;
```

```
  final String addressLine1;
  final String addressCity;
  final String addressState;
  final List<Person> children;

  const Person(this.name, this.addressLine1, this.addressCity,
      this.addressState, this.children);

  // You write this serialization code.
  Map<String, dynamic> toJson() {
    var map = {
      'name': name,
      'addr': addressLine1,
      'city': addressCity,
      'state': addressState,
      'children': children
    };
    return map;
  }
  // You write this serialization code.

  @override
  String toString() {
    return 'Person{name: $name, addressLine1: $addressLine1,
addressCity: $addressCity, addressState: $addressState,
children: $children}';
  }
}
```

```
class MyApp extends StatelessWidget {
  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Demo',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: HomeWidget(),
    );
  }
}

class HomeWidget extends StatelessWidget {
  static const Person _grandchild =
      Person("Tracy Brown", "9625 Roberts Avenue",
"Birmingham", "AL", []);
  static const Person _adultFather = const Person(
      "John Brown", "9625 Roberts Avenue", "Birmingham",
"AL", [_grandchild]);
  static const Person _adultNoChildren =
      const Person("Jill Jones", "100 East Road", "Ocala", "FL",
[]);
  static const Person _grandfather = Person("John Brown",
"9621 Roberts Avenue",
      "Birmingham", "AL", [_adultFather, _adultNoChildren]);
```

```
  @override
  Widget build(BuildContext context) {
    return Scaffold(
        appBar: AppBar(
          title: Text("Recursive Serialization"),
        ),
        body: Center(
          child: Padding(
            child: ListView(
              children: <Widget>[
                Padding(
                    child: Text("Grandfather:\n${_grandfather}"),
                    padding: EdgeInsets.only(top: 0.0)),
                Padding(
                    child: Text("Json
Encoded:\n${json.encode(_grandfather)}",
                        style: TextStyle(color: Colors.red)),
                    padding: EdgeInsets.only(top: 10.0)),
                FlatButton(
                    child: Text("Copy"),
                    onPressed: (() {
                      Clipboard.setData(
                          ClipboardData(text:
"${json.encode(_grandfather)}"));
                    })),
                Padding(
                    child: Text("Adult Father:\n${_adultFather}"),
```

```
                padding: EdgeInsets.only(top: 30.0)),
            Padding(
                child: Text("Json
Encoded:\n${json.encode(_adultFather)}",
                    style: TextStyle(color: Colors.red)),
                padding: EdgeInsets.only(top: 10.0)),
            FlatButton(
                child: Text("Copy"),
                onPressed: (() {
                  Clipboard.setData(
                      ClipboardData(text:
"${json.encode(_adultFather)}"));
                })),
            Padding(
                child: Text("Adult No
Children:\n${_adultNoChildren}"),
                padding: EdgeInsets.only(top: 30.0)),
            Padding(
                child: Text(
                    "Json
Encoded:\n${json.encode(_adultNoChildren)}",
                    style: TextStyle(color: Colors.red)),
                padding: EdgeInsets.only(top: 10.0)),
            FlatButton(
                child: Text("Copy"),
                onPressed: (() {
                  Clipboard.setData(ClipboardData(
```

```
                    text: "${json.encode(_adultNoChildren)}"));
                })),
            Padding(
                child: Text("Grandchild:\n${_grandchild}"),
                padding: EdgeInsets.only(top: 30.0)),
            Padding(
                child: Text("Json
Encoded:\n${json.encode(_grandchild)}",
                    style: TextStyle(color: Colors.red)),
                padding: EdgeInsets.only(top: 10.0)),
            FlatButton(
                child: Text("Copy"),
                onPressed: (() {
                  Clipboard.setData(
                      ClipboardData(text:
"${json.encode(_grandchild)}"));
                })),
          ],
        ),
        padding: EdgeInsets.all(10.0),
      ),
    ));
  }
}
```

Example – ‘deserialize_manually’

This app lets you enter the JSON for a person then hit

the floating button to deserialize it.

- If successful, a ‘toString()’ of the Person object is displayed underneath (in black).
- If an error occurs (maybe you input bad JSON?), it is displayed underneath (in red).

Remember that this should work recursively - the Person JSON can have children, which will create a Person object with children (and so on). This app defaults your initial JSON input to the grandparent John Brown so that you can see this recursion working.

This app also writes to the console so you can follow whats happening.


Recursive Deserialization
Enter the json for a person
{"name":"John Brown", "addr1":"9621 Roberts Avenue","city":"Birmingham","state":"AL", "children": [{"name":"John Brown", "addr1":"9625 Roberts Avenue","city":"Birmingham","state":"AL", "children": [{"name":"Tracy Brown", "addr1":"9625 Roberts Avenue","city":"Birmingham","state":"AL", "children": []}]},{"name":"Jill Jones", "addr1":"100 East Road","city":"Ocala", "state":"FL", "children":[]}]}
Converted to Person object:
Person{name: John Brown, addressLine1: 9621 Roberts Avenue, addressCity: Birmingham, addressState: AL, children: [Person{name: John Brown, addressLine1: 9625 Roberts Avenue, addressCity: Birmingham, addressState: AL, children: [Person{name: Tracy Brown, addressLine1: 9625 Roberts Avenue, addressCity: Birmingham, addressState: AL, children: []}], Person{name: Jill Jones, addressLine1: 100 East Road, addressCity: Ocala, addressState: FL, children: []}}

Source Code

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

void main() => runApp(MyApp());

class Person {
  final String name;
  final String addressLine1;
```

```
  final String addressCity;
  final String addressState;
  final List<Person> children;

  const Person(this.name, this.addressLine1, this.addressCity,
      this.addressState, this.children);

  // You write this deserialization code.
  factory Person.fromJson(Map<String, dynamic> json) {
    if (json == null) {
      throw FormatException("Null JSON.");
    }

    // Recursion. Convert children into list of Person objects.
    List<dynamic> decodedChildren = json['children'];
    List<Person> children = [];
    decodedChildren.forEach((decodedChild) {
      children.add(Person.fromJson(decodedChild));
    });

    return Person(
        json['name'], json['addr1'], json['city'], json['state'],
children);
  }
  // You write this deserialization code.

  @override
  String toString() {
```

```
    return 'Person{name: $name, addressLine1: $addressLine1,
addressCity: $addressCity, addressState: $addressState,
children: $children}';
  }
}

class MyApp extends StatelessWidget {
  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Demo',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: HomeWidget(),
    );
  }
}

class HomeWidget extends StatefulWidget {
  HomeWidget({Key key}) : super(key: key);

  @override
  _HomeWidgetState createState() => _HomeWidgetState();
}

class _HomeWidgetState extends State<HomeWidget> {
```

```
  final _jsonTextController = TextEditingController();
  Person _person;
  String _error;

  _HomeWidgetState() {
    final String grandchild =
        "{\"name\":\"Tracy Brown\", \"addr1\":\"9625 Roberts
Avenue\"," +
            "\"city\":\"Birmingham\", \"state\":\"AL\",
\"children\":[" +
            "]}";
    final String adultFather =
        "{\"name\":\"John Brown\", \"addr1\":\"9625 Roberts
Avenue\"," +
            "\"city\":\"Birmingham\", \"state\":\"AL\",
\"children\":[" +
            grandchild +
            "]}";
    final String adultNoChildren =
        "{\"name\":\"Jill Jones\", \"addr1\":\"100 East Road\","
+
            "\"city\":\"Ocala\", \"state\":\"FL\", \"children\":[" +
            "]}";
    final String grandfather =
        "{\"name\":\"John Brown\", \"addr1\":\"9621 Roberts
Avenue\"," +
            "\"city\":\"Birmingham\", \"state\":\"AL\",
\"children\":[" +
```

```
        adultFather +
        "," +
        adultNoChildren +
        "]}";

  _jsonTextController.text = grandfather;
}

TextFormField _createJsonTextFormField() {
  return new TextFormField(
      validator: (value) {
        if (value.isEmpty) {
          return 'Please enter the json.';
        }
      },
      decoration: InputDecoration(
          border: OutlineInputBorder(),
          hintText: 'Json',
          labelText: 'Enter the json for a person.'),
      controller: _jsonTextController,
      autofocus: true,
      maxLines: 8,
      keyboardType: TextInputType.multiline);
}

_convertJsonToPerson() {
  _error = null;
```

```
    _person = null;
    setState(() {
      try {
        final String jsonText = _jsonTextController.text;
        debugPrint("JSON TEXT: ${jsonText}");
        var decoded = json.decode(jsonText); // text to map
        debugPrint("DECODED:  type: ${decoded.runtimeType}
value: ${decoded}");
        _person = Person.fromJson(decoded); // map to object
        debugPrint("PERSON OBJECT: type:
${_person.runtimeType} value: "
            "${_person}");
      } catch (e) {
        debugPrint("ERROR: ${e}");
        _error = e.toString();
      }
    });
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text("Recursive Deserialization"),
      ),
      body: Center(
        child: Padding(
```

```
        child: ListView(
          children: <Widget>[
            _createJsonTextFormField(),
            Padding(
                padding: EdgeInsets.only(top: 10.0),
                child: Text(
                    _error == null ? '' : 'An error
occurred:\n\n${_error}',
                    style: TextStyle(color: Colors.red))),
            Padding(
                padding: EdgeInsets.only(top: 10.0),
                child: Text(_person == null
                    ? 'Person is null'
                    : 'Converted to Person object:\n\n${_person}'))
          ],
        ),
        padding: EdgeInsets.all(10.0),
      ),
    ),
    floatingActionButton: FloatingActionButton(
      onPressed: _convertJsonToPerson,
      tooltip: 'Increment',
      child: Icon(Icons.refresh),
    ), // This trailing comma makes auto-formatting nicer for
build methods.
  );
 }
```

```
}
```

Flutter & HTTP

Introduction

Now we know how to convert the data from Flutter to JSON and back again, we need to write code that communicates with APIs on servers, using the HTTP protocol.

Flutter HTTP Package

To do this we will use the Flutter HTTP Package. It is not a core package so we will have to add a dependency for it.

Dependency

To use it, you have to add the dependency to your project in the ‘pubspec.yaml’ file:

```
dependencies:
  HTTP: ^0.12.0+1
```

Remember to do a ‘flutter packages get’ afterwards.

More info here: https://pub.dartlang.org/packages/http

Dummy API

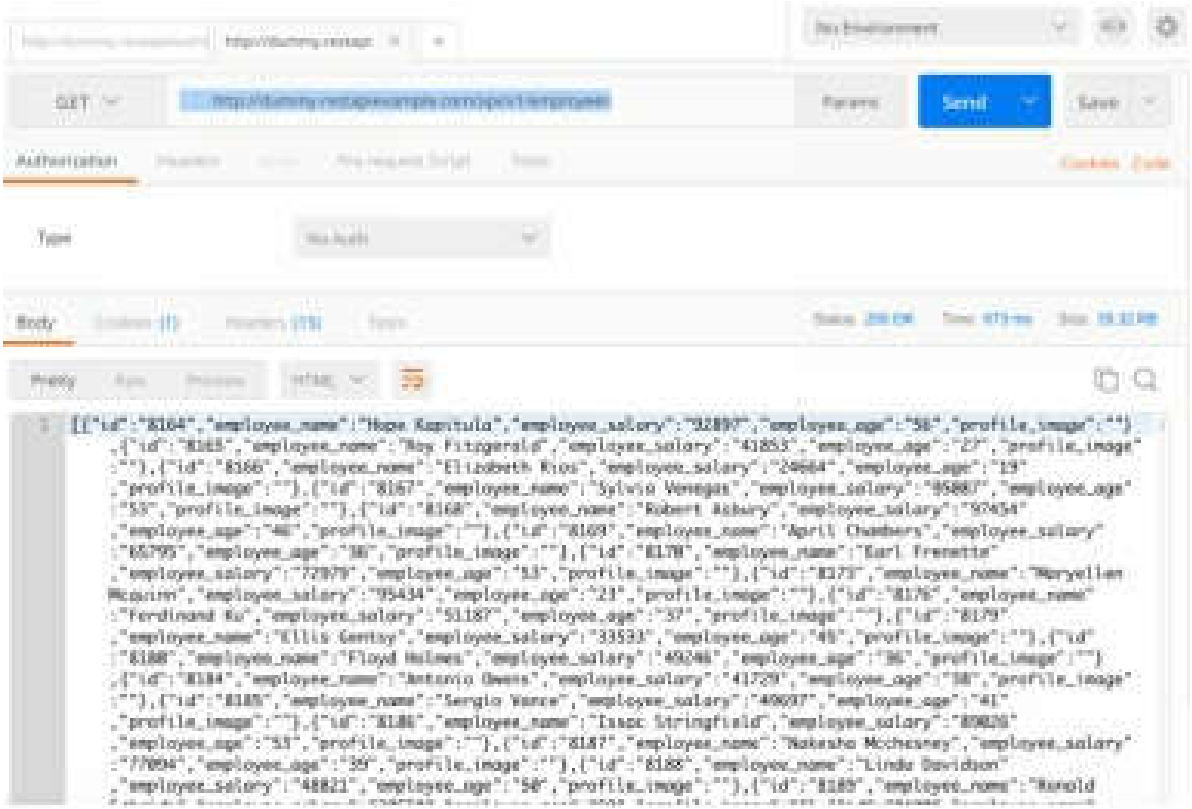
We are going to use someone’s API for these exercises

and for the example code.
In this case we are going to use the dummy rest api here: http://dummy.restapiexample.com/, because it covers all of the following: get data, add data, update data and delete data. It also doesn’t require a key or registration.

Exercise - Get Data Using Postman

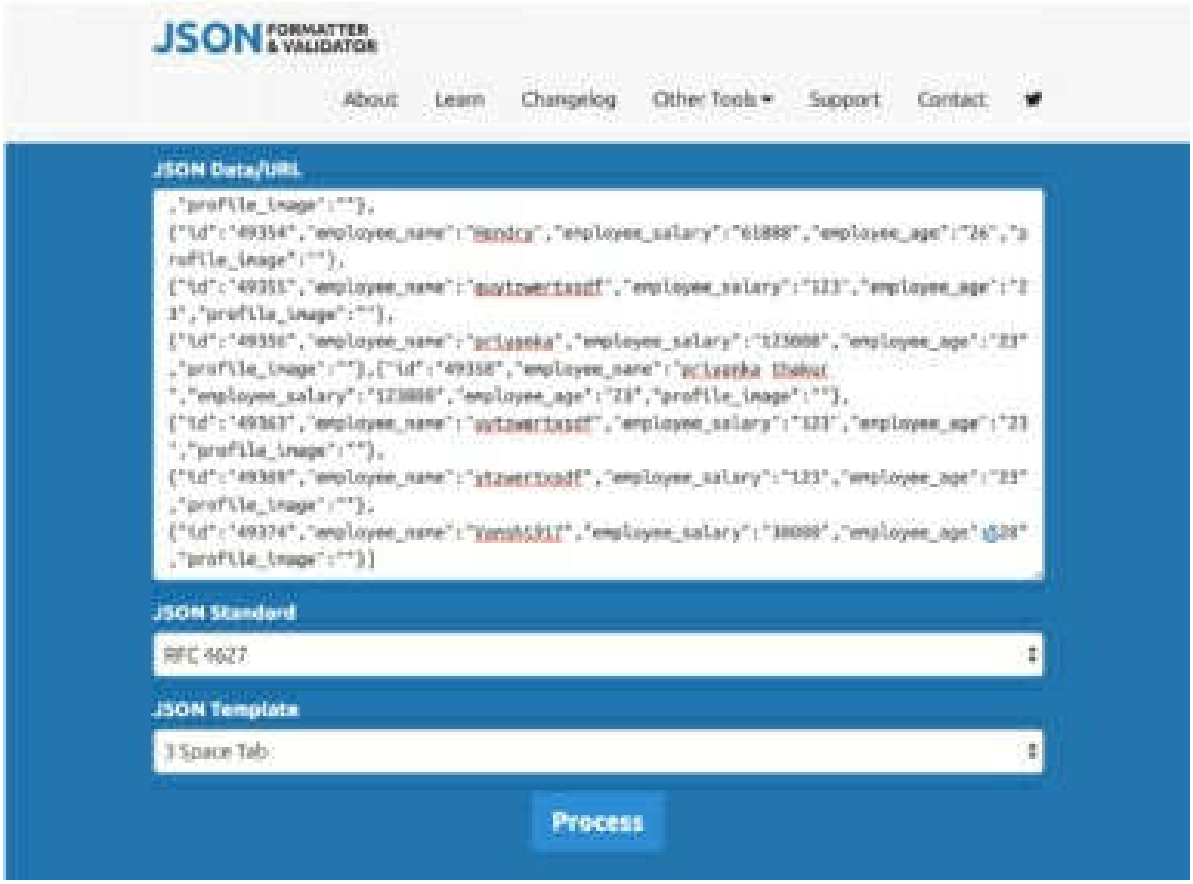
In this exercise, we will use the API to get information about employees.

- Open Postman
- Copy and paste 'http://dummy.restapiexample.com/api/v1/employ into the Request URL at top.
- Hit the ‘Send’ button.
- Data should show up at the bottom.

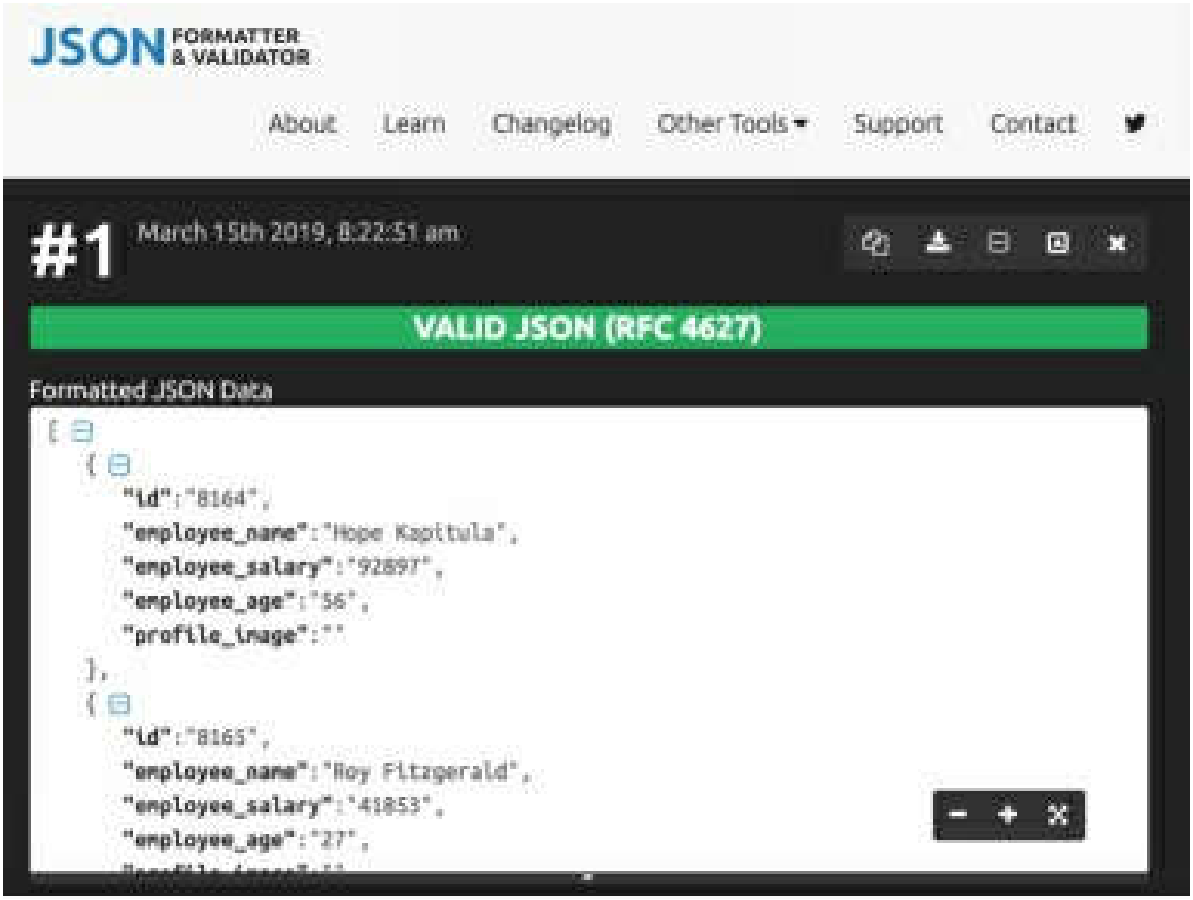


Exercise – Format Data

- Click on the data near the bottom, then select all and copy.
- Go to https://jsonformatter.curiousconcept.com/ in your browser.
- Paste the data into the box ‘JSON Data/URL’ (see below).



- Hit the ‘Process’ button. You should be taken to a formatted view of the data (see below).



Error Handling

As mentioned in the chapter before, the Flutter HTTP package enables us to communicate with APIs asynchronously using HTTP and this makes error handling a little more complex:

- You need to add an error handler in case an error occurs when you first make the request.
- You need to add an error handler incase the future terminates with an error.
- You need to check the HTTP code of the

response from the server incase anything was incorrect or went wrong on that end.

Please refer to the error handling in the example code below.

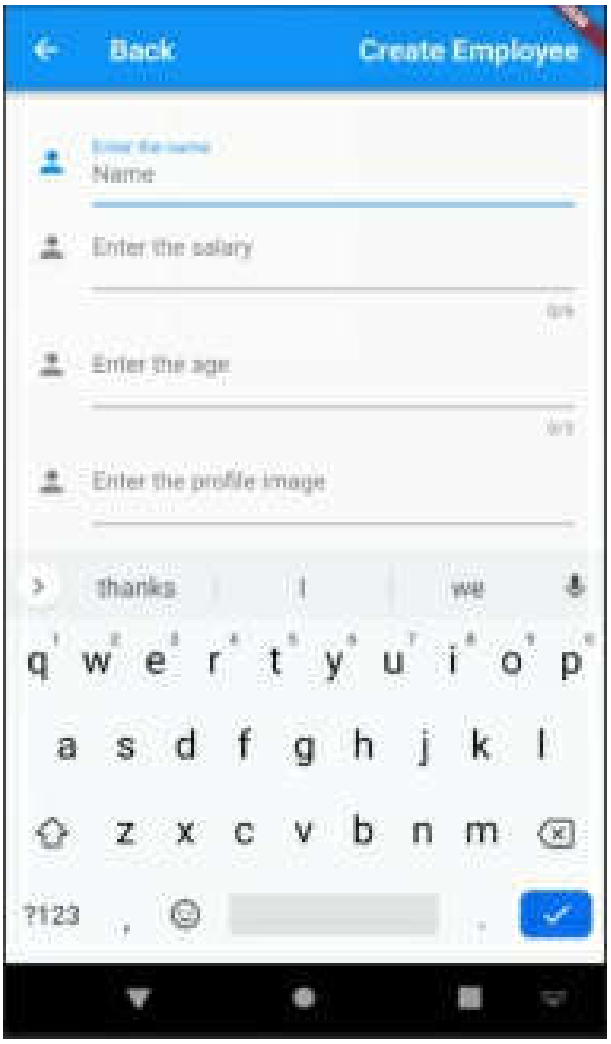
Example ‘http_employees’

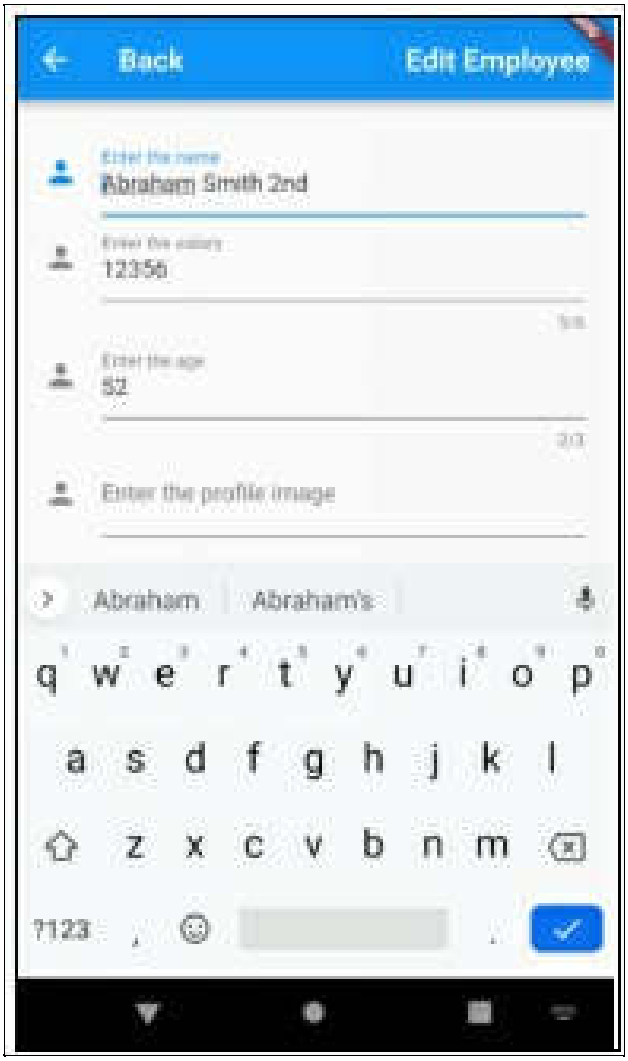
In preparing this example, I had to find an API that was public to work with that would work with all of the HTTP verbs, so you could see getting data, adding data, updating data and deleting data. I ended up using http://dummy.restapiexample.com/ . It is a REST Api that enables people to maintain a list of employees. Like many such Apis, does not exactly subscribe to the REST pattern prescribed in this chapter. Some of the url patterns have been interpreted differently to how I expected them to. However, it is good to use for an example and I am grateful to them for putting it out there.

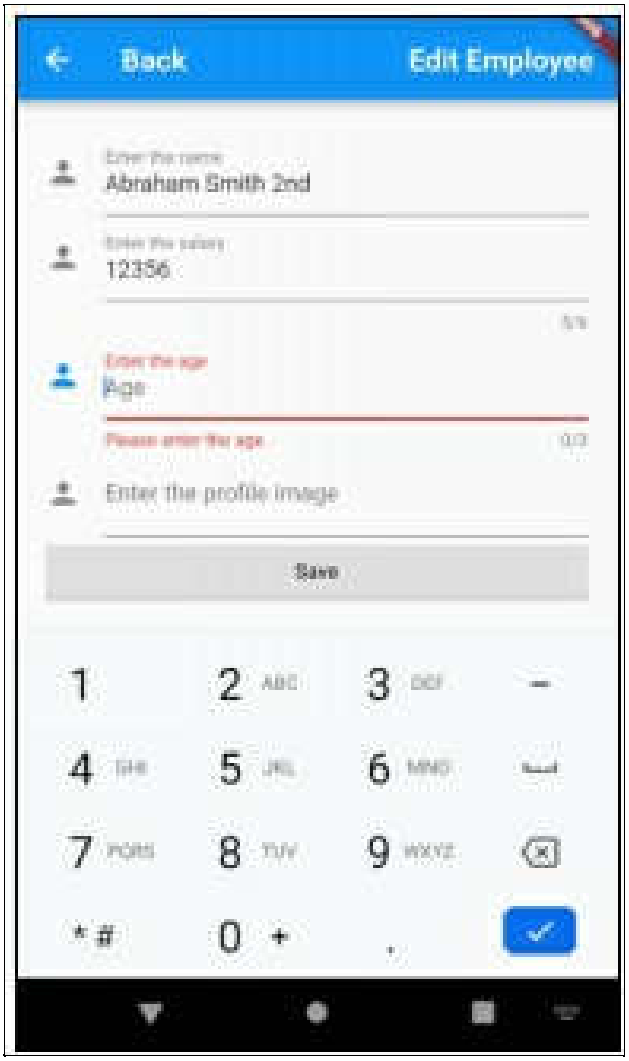
This example app connects to dummy Api and enables you to add employees, update employees and delete them. It starts with a list of employees and you can tap on one to view and make changes. You can also delete employees but tapping longer on an employee in the list of employees.

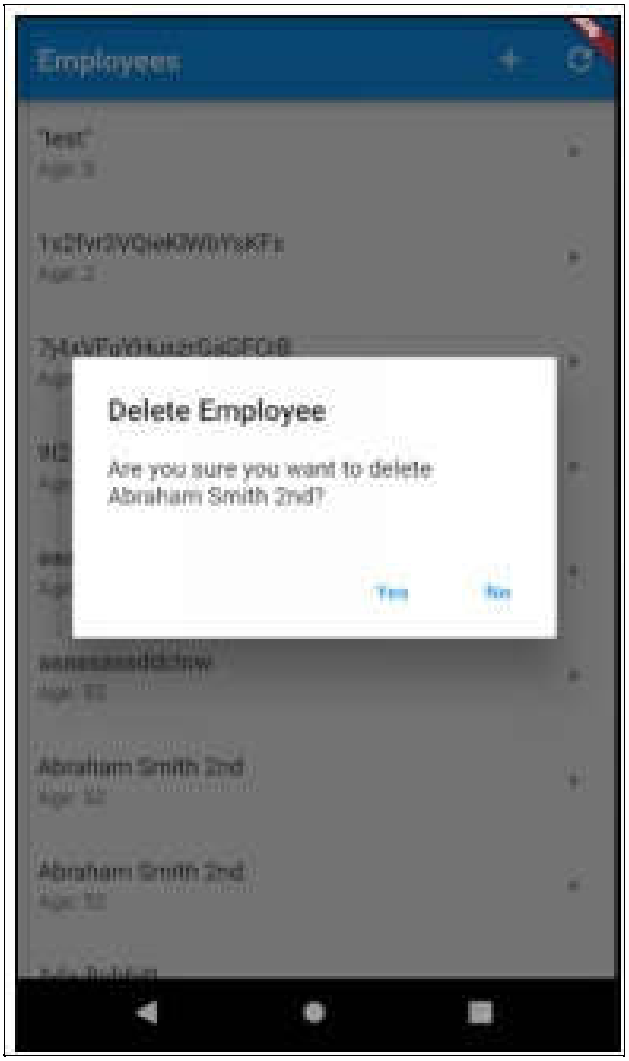
This example app should also demonstrate how you may sometimes encounter errors when communicating with Apis. For example, the dummy Api doesn’t allow the same employee name twice. If you enter the same employee name twice and attempt to save, then the dummy Api will return an error and this is displayed to the user. This could be handled more gracefully but at least it catches it and shows some information at the bottom.


Employees
'test'
Age: 0
1s2fvr3VQieK3WbYsKFs
Age: 2
7j4xVFuYHuxznGaGFCnB
Age: 12
9t21Q7xH6IzAtaJBZUpc
Age: 22
aaaaa
Age: 22
aaaasaassddchw
Age: 92
Abraham Smith 2nd
Age: 92
Abraham Smith 2nd
Age: 92


Back
Create Employee
Name
Enter the salary
Enter the age
Enter the profile image
thanks
I
we
q w e r t y u i o p
a s d f g h j k l
z x c v b n m
?123


Back
Edit Employee
Abraham Smith 2nd
12356
52
Enter the profile image
Abraham
Abraham's


Back
Edit Employee
Abraham Smith 2nd
12356
Age
Enter the profile image
Save



This example app may be useful because it combines multiple Flutter topics together:

- Communicating with a REST Api on an HTTP server.
- Forms and validation.
- Modal dialogs.
- State management using inherited widget and stateful widgets.

- Error handling.

Source Code

Dependencies

Add the following dependencies into the ‘pubspec.yaml’ file. After that you will need to do a ‘flutter packages get’ on the command line in the root of your project to download the dependencies.

```
dependencies:
  flutter:
    sdk: flutter
  rxdart: 0.18.1
  HTTP: ^0.11.0
  cupertino_icons: ^0.1.2
```

Source Code

```
import 'dart:async';
import 'dart:convert';

import 'package:flutter/material.dart';
import 'package:flutter/services.dart';
import 'package:HTTP/HTTP.dart' as HTTP;
import 'package:HTTP/HTTP.dart';

void main() => runApp(new MyApp());

class Employee {
```

```
  String id;
  String employeeName;
  String employeeSalary;
  String employeeAge;
  String profileImage;

  Employee(this.id, this.employeeName, this.employeeSalary,
this.employeeAge,
      this.profileImage);

  Employee.empty() {
    id = "";
    employeeName = "";
    employeeSalary = "";
    employeeAge = "";
    profileImage = "";
  }

  factory Employee.fromJson(Map<String, dynamic> json) {
    if (json == null) {
      throw FormatException("Null JSON.");
    }
    return Employee(json['id'], json['employee_name'],
json['employee_salary'],
        json['employee_age'], json['profile_image']);
  }

  Map<String, dynamic> toJson() {
```

```
    var map = {
      'name': employeeName,
      'salary': employeeSalary,
      'age': employeeAge
    };
    if (id.isNotEmpty) {
      map['id'] = id;
    }
    if (profileImage.isNotEmpty) {
      map['profileImage'] = profileImage;
    }
    return map;
  }

  get hasEmptyId {
    return id.isEmpty;
  }
}

class PleaseWaitWidget extends StatelessWidget {
  PleaseWaitWidget({
    Key key,
  }) : super(key: key);

  // This widget is the root of your application.
  @override
  Widget build(BuildContext context) {
```

```
    return Container(
        child: Center(
          child: CircularProgressIndicator(),
        ),
        color: Colors.white.withOpacity(0.8));
  }
}

class ApiWidget extends InheritedWidget {
  static final String _BASE_URL =
"http://dummy.restapiexample.com/api/v1";
  static const _TIMEOUT = Duration(seconds: 10);

  ApiWidget({
    Key key,
    @required Widget child,
  })  : assert(child != null),
        super(key: key, child: child);

  static ApiWidget of(BuildContext context) {
    return context.inheritFromWidgetOfExactType(ApiWidget)
as ApiWidget;
  }

  @override
  bool updateShouldNotify(covariant InheritedWidget
oldWidget) {
    return false;
```

```
  }

  Future<List<Employee>> loadAndParseEmployees() async {
    var url = '${_BASE_URL}/employees';
    final response = await HTTP.get(url).timeout(_TIMEOUT);
    if (response.statusCode == 200) {
      final parsed =
json.decode(response.body).cast<Map<String, dynamic>>();
      var list =
          parsed.map<Employee>((json) =>
Employee.fromJson(json)).toList();
      return list;
    } else {
      badStatusCode(response);
    }
  }

  Future<Employee> loadEmployee(String id) async {
    var url = '${_BASE_URL}/employee/${id}';
    final response = await HTTP.get(url).timeout(_TIMEOUT);
    if (response.statusCode == 200) {
      final parsed = json.decode(response.body);
      return Employee.fromJson(parsed);
    } else {
      badStatusCode(response);
    }
  }
```

```
  Future<dynamic> saveEmployee(Employee employee) async
{
    bool isUpdate = employee.id.isNotEmpty;
    final uri = _BASE_URL + (isUpdate ?
'/update/${employee.id}' : '/create');
    // profile image does not seem to update
    final response = isUpdate
        ? await HTTP.put(uri, body:
json.encode(employee)).timeout(_TIMEOUT)
        : await HTTP.post(uri, body:
json.encode(employee)).timeout(_TIMEOUT);
    if (response.statusCode == 200) {
      return json.decode(response.body);
    } else {
      // If that response was not OK, throw an error.
      badStatusCode(response);
    }
  }

  Future<dynamic> deleteEmployee(String id) async {
    final uri = '${_BASE_URL}/delete/${id}';
    final response = await
HTTP.delete(uri).timeout(_TIMEOUT);
    if (response.statusCode == 200) {
      return json.decode(response.body);
    } else {
      // If that response was not OK, throw an error.
      badStatusCode(response);
```

```
    }
  }

  badStatusCode(Response response) {
    debugPrint("Bad status code ${response.statusCode} returned from server.");
    debugPrint("Response body ${response.body} returned from server.");
    throw Exception(
        'Bad status code ${response.statusCode} returned from server.');
  }
}

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return new ApiWidget(
        child: MaterialApp(
            title: 'Flutter Demo',
            theme: new ThemeData(
              primarySwatch: Colors.blue,
            ),
            home: new EmployeeListWidget()));
  }
}

class EmployeeListWidget extends StatefulWidget {
```

```
  @override
  _EmployeeListWidgetState createState() => new
_EmployeeListWidgetState();
}

class _EmployeeListWidgetState extends
State<EmployeeListWidget> {
  final GlobalKey<ScaffoldState> _scaffoldKey =
GlobalKey<ScaffoldState>();
  final PleaseWaitWidget _pleaseWaitWidget =
      PleaseWaitWidget(key: ObjectKey("pleaseWaitWidget"));

  bool _refresh = true;
  List<Employee> _employees;
  bool _pleaseWait = false;

  _showSnackBar(String content, {bool error = false}) {
    _scaffoldKey.currentState.showSnackBar(SnackBar(
      content:
          Text('${error ? "An unexpected error occurred: " :
""}${content}'),
    ));
  }

  _showPleaseWait(bool b) {
    setState(() {
      _pleaseWait = b;
    });
  }
```

```
  _navigateToEmployee(BuildContext context, String
employeeId) {
    Navigator.push(
      context,
      MaterialPageRoute(builder: (context) =>
EmployeeDetailWidget(employeeId)),
    ).then((result) {
      if ((result != null) && (result is bool) && (result ==
true)) {
        _showSnackBar('Employee saved.');
        _refreshEmployees();
      }
    });
  }

  _deleteEmployee(BuildContext context, Employee employee)
async {
    _showDeleteConfirmDialog(employee).then((result) {
      if ((result != null) && (result is bool) && (result ==
true)) {
        _showPleaseWait(true);
        try {
          ApiWidget.of(context).deleteEmployee(employee.id).the
{
            _showPleaseWait(false);
            _showSnackBar('Employee deleted.');
            _refreshEmployees();
          }).catchError((error) {
```

```
          _showPleaseWait(false);
          _showSnackBar(error.toString(), error: true);
        });
      } catch (e) {
        _showPleaseWait(false);
        _showSnackBar(e.toString(), error: true);
      }
    }
  });
}

Future<bool> _showDeleteConfirmDialog(Employee
employee) async {
  return await showDialog<bool>(
      context: context,
      builder: (BuildContext context) {
        return AlertDialog(
          title: const Text('Delete Employee'),
          content: Text(
              'Are you sure you want to delete
${employee.employeeName}?'),
          actions: <Widget>[
            FlatButton(
              onPressed: () {
                Navigator.pop(context, true);
              },
              child: const Text('Yes'),
```

```
            ),
            FlatButton(
              onPressed: () {
                Navigator.pop(context, false);
              },
              child: const Text('No'),
            )
          ],
        );
      });
  }

  _refreshEmployees() {
    setState(() {
      _refresh = true;
    });
  }

  _loadEmployees(BuildContext context) {
    _showPleaseWait(true);
    try {
      ApiWidget.of(context).loadAndParseEmployees().then((emp
{
        // Sort first.
        employees.sort((a, b) => a.employeeName
            .toLowerCase()
            .compareTo(b.employeeName.toLowerCase()));
```

```
      setState(() {
        _employees = employees;
      });
      _showPleaseWait(false);
    }).catchError((error) {
      _showPleaseWait(false);
      _showSnackBar(error.toString(), error: true);
    });
  } catch (e) {
    _showPleaseWait(false);
    _showSnackBar(e.toString(), error: true);
  }
}

@override
Widget build(BuildContext context) {
  if (_refresh) {
    _refresh = false;
    _loadEmployees(context);
  }

  ListView builder = ListView.builder(
      itemCount: _employees != null ? _employees.length : 0,
      itemBuilder: (context, index) {
        Employee employee = _employees[index];
        return ListTile(
            title: Text('${employee.employeeName}'),
```

```
            subtitle: Text('Age: ${employee.employeeAge}'),
            trailing: Icon(Icons.arrow_right),
            onTap: () => _navigateToEmployee(context,
employee.id),
            onLongPress: () => _deleteEmployee(context,
employee));
        });

    Widget bodyWidget = _pleaseWait
        ? Stack(key: ObjectKey("stack"), children:
[_pleaseWaitWidget, builder])
        : Stack(key: ObjectKey("stack"), children: [builder]);

    return new Scaffold(
        key: _scaffoldKey,
        appBar: new AppBar(
          title: new Text("Employees"),
          actions: <Widget>[
            IconButton(
                icon: Icon(Icons.add),
                tooltip: 'Add',
                onPressed: () {
                  _navigateToEmployee(context, null);
                }),
            IconButton(
                icon: Icon(Icons.refresh),
                tooltip: 'Refresh',
                onPressed: () {
```

```
                _refreshEmployees();
              })
          ],
        ),
        body: new Center(
          child: bodyWidget,
        ));
  }
}

class EmployeeDetailWidget extends StatefulWidget {
  String _employeeId;

  EmployeeDetailWidget(this._employeeId);

  @override
  _EmployeeDetailState createState() =>
_EmployeeDetailState(this._employeeId);
}

class _EmployeeDetailState extends
State<EmployeeDetailWidget> {
  final GlobalKey<ScaffoldState> _scaffoldKey =
GlobalKey<ScaffoldState>();
  final _formKey = GlobalKey<FormState>();
  final PleaseWaitWidget _pleaseWaitWidget =
      PleaseWaitWidget(key: ObjectKey("pleaseWaitWidget"));

  String _employeeId;
```

```
  bool _loaded = false;
  bool _pleaseWait = false;
  Employee _employee;
  TextEditingController _nameTextController =
TextEditingController();
  TextEditingController _salaryTextController =
TextEditingController();
  TextEditingController _ageTextController =
TextEditingController();
  TextEditingController _profileImageTextController =
TextEditingController();

  _EmployeeDetailState(this._employeeId);

  _showSnackBar(String content, {bool error = false}) {
    _scaffoldKey.currentState.showSnackBar(SnackBar(
      content:
          Text('${error ? "An unexpected error occurred: " :
""}${content}'),
    ));
  }

  _showPleaseWait(bool b) {
    setState(() {
      _pleaseWait = b;
    });
  }

  TextFormField _createNameWidget() {
```

```
  return new TextFormField(
    validator: (value) {
      if (value.isEmpty) {
        return 'Please enter the name.';
      }
    },
    decoration: InputDecoration(
        icon: const Icon(Icons.person),
        hintText: 'Name',
        labelText: 'Enter the name'),
    onSaved: (String value) {
      this._employee.employeeName = value;
    },
    controller: _nameTextController,
    autofocus: true,
  );
}

TextFormField _createSalaryWidget() {
  return new TextFormField(
    validator: (value) {
      if (value.isEmpty) {
        return 'Please enter the salary.';
      }
      int salary = int.parse(value);
      if (salary == null) {
```

```
        return 'Please enter the salary as a number.';
      }
      if ((salary < 10000) || (salary > 500000)) {
        return 'Please enter an age between 10000 and 50000.';
      }
    },
    maxLength: 6,
    maxLengthEnforced: true,
    keyboardType: TextInputType.phone,
    inputFormatters:
[WhitelistingTextInputFormatter.digitsOnly],
    decoration: InputDecoration(
        icon: const Icon(Icons.person),
        hintText: 'Salary',
        labelText: 'Enter the salary'),
    onSaved: (String value) {
      this._employee.employeeSalary = value;
    },
    controller: _salaryTextController,
  );
 }

 TextFormField _createAgeWidget() {
  return new TextFormField(
    validator: (value) {
      if (value.isEmpty) {
        return 'Please enter the age.';
```

```
        }
        int age = int.parse(value);
        if (age == null) {
          return 'Please enter the age as a number.';
        }
        if ((age < 1) || (age > 114)) {
          return 'Please enter an age between 1 and 114.';
        }
      },
      maxLength: 3,
      maxLengthEnforced: true,
      keyboardType: TextInputType.phone,
      inputFormatters:
[WhitelistingTextInputFormatter.digitsOnly],
      decoration: InputDecoration(
          icon: const Icon(Icons.person),
          hintText: 'Age',
          labelText: 'Enter the age'),
      onSaved: (String value) {
        this._employee.employeeAge = value;
      },
      controller: _ageTextController,
    );
  }

  TextFormField _createProfileImageWidget() {
    return new TextFormField(
```

```
      decoration: InputDecoration(
          icon: const Icon(Icons.person),
          hintText: 'Profile image',
          labelText: 'Enter the profile image'),
      onSaved: (String value) {
        this._employee.profileImage = value;
      },
      controller: _profileImageTextController,
    );
  }

  _loadEmployee(BuildContext context) {
    _showPleaseWait(true);
    try {
      ApiWidget.of(context).loadEmployee(_employeeId).then((e
{
        setState(() {
          _employee = employee;
          _nameTextController.text = employee.employeeName;
          _salaryTextController.text = employee.employeeSalary;
          _ageTextController.text = employee.employeeAge;
          _profileImageTextController.text =
employee.profileImage;
        });
        _showPleaseWait(false);
      }).catchError((error) {
        _showPleaseWait(false);
```

```
        _showSnackBar(error.toString(), error: true);
      });
    } catch (e) {
      _showPleaseWait(false);
      _showSnackBar(e.toString(), error: true);
    }
  }

  _saveEmployee(BuildContext context) {
    _showPleaseWait(true);
    try {
      ApiWidget.of(context).saveEmployee(_employee).then((em
{
        _showPleaseWait(false);
        Navigator.pop(context, true);
      }).catchError((error) {
        _showPleaseWait(false);
        _showSnackBar(error.toString(), error: true);
      });
    } catch (e) {
      _showPleaseWait(false);
      _showSnackBar(e.toString(), error: true);
    }
  }

  @override
  Widget build(BuildContext context) {
```

```
    if (!_loaded) {
      _loaded = true;
      if (_employeeId == null) {
        _employee = Employee.empty();
      } else {
        _loadEmployee(context);
      }
    }

    List<Widget> formWidgetList = [
      _createNameWidget(),
      _createSalaryWidget(),
      _createAgeWidget(),
      _createProfileImageWidget(),
      RaisedButton(
        onPressed: () {
          if (_formKey.currentState.validate()) {
            _formKey.currentState.save();
            _saveEmployee(context);
          }
        },
        child: new Text('Save'),
      )
    ];
    Form form = Form(key: _formKey, child:
ListView(children: formWidgetList));
```

```
    Widget bodyWidget = _pleaseWait
        ? Stack(key: ObjectKey("stack"), children:
[_pleaseWaitWidget, form])
        : Stack(key: ObjectKey("stack"), children: [form]);

    return new Scaffold(
        key: _scaffoldKey,
        appBar: new AppBar(
          title: new Row(children: [
            Text("Back"),
            Spacer(),
            Text(_employeeId == null ? "Create Employee" :
"Edit Employee")
          ]),
        ),
        body: new Padding(padding: EdgeInsets.all(20.0), child:
bodyWidget));
  }
}
```

Other Information

Alice

One of the useful things about doing web development is that your web browser has a ‘developer tools’ console that lets you inspect the HTTP traffic. Unfortunately, your app does not have this built in.

Alice is a package that can use to inspect the HTTP traffic going between your app and HTTP servers. It has turned out to be both easy to use and useful to me.

Further Reading

Https://medium.com/flutter-community/inspecting-HTTP-requests-in-flutter-9deeddfe8d1

HAL / HATEOS

To talk to the server, apps need to know the URLs that the server resources are available on. Most of the time this information is hardcoded, which is not ideal.

It is much better if the server tells incudes information about available resources (and their URLs) when it returns information in the response back to the app. There are various standards as to the format of sending this information back to the client, including HATEOS & HAL.

For example, if you have an app which sends a request to the server to retrieve a list of customers, the information could should include the URLs for the API calls to access the data for each customer. This avoids hardcoding the customer AJAX request URL.

Further Reading

Https://martinfowler.com/articles/richardsonMaturityM
Https://en.wikipedia.org/wiki/HATEOAS

28. State

Introduction

So now we know the basics about Widgets, composition and how we can get data from servers, we need to start writing interactive apps. However, to write interactive apps you first need to consider state and events.

The purpose of this chapter is to introduce state and events.

State & Events

State is the data in the app, often displayed in the UI.
Events are what may happen in the app.
You want Events to affect State, that’s an Interactive User Interface is all about.

Storing State

Say we have an application structured like this:

- The user logs into the app using their username and password in a Login Widget, which talks to a server.
- The server gets the username and password info from the Login Widget and returns info about the

user.

- The user enters data in a Data Entry Widget. This widget needs info about the user, i.e. what kinds of data entry can be performed by the user.
- The user views reports in a Report Widget. This widget needs info about the user, i.e. what reports can be viewed by the user.

Note the locations of the state in the diagram below (white text with grey background).

Kinds of State

In this example, there are 2 kinds of state.

- Local State – this is state info which is just needed in one place.
    - For example, the username and password are needed in the Login Widget but in no other widgets.

- Global State – this is state info which is needed almost everywhere.
    - For example, the user info is needed in multiple widgets, to know what kinds of data entry the user can do and/or what reports he or she can view.

How to Determine Where to Store State

These points are just a guideline:

1. Remember the golden rule - keep things simple.
2. Don’t store state unnecessarily. Store what you absolutely need to store as state and no more.
3. Don’t repeat state. Don’t store the same item multiple times in state.

See that the user info state is stored up in the App Stateful Widget, above the Data Entry and Report Widgets? It was moved up a level in the object hierarchy so that it’s not repeated. The child widgets can get that state info from their parent widget.

4. Place the state as close to where it is needed.

See that the username and password state are stored in

the Login Stateful Widget. That is because user username and password state is local as its not needed anywhere else.

Responding to Events

Introduction

A modern user interface reacts to Events:

- User clicking on buttons.
- Data coming in from a server.
- Time passing.
- Etc

Events Invoke Functions

When you add your code to handle events, you typically assign a function to the event handler. The function that has code to do something, usually affect State.
You can assign regular functions to events or you can assign arrow functions instead.

```
@override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
```

```
        title: Text(widget.title),
      ),
      body: Column(
        children: <Widget>[
          Text("Do you want to\nbuy this item?"),
          Row(
            mainAxisAlignment: MainAxisAlignment.center,
            children: <Widget>[
              Spacer(flex: 2),
              SelectButton(text: "YES", onTap: _yesOnTap), /*
regular function */
              Spacer(),
              SelectButton(text: "NO", onTap: () => print('no') /*
arrow function */),
              Spacer(flex: 2),
            ],
          )
        ]),
    );
  }

  void _yesOnTap() {
    print('yes');
  }
```

Events Can Affect State

When Events occur, they invoke code that tends to

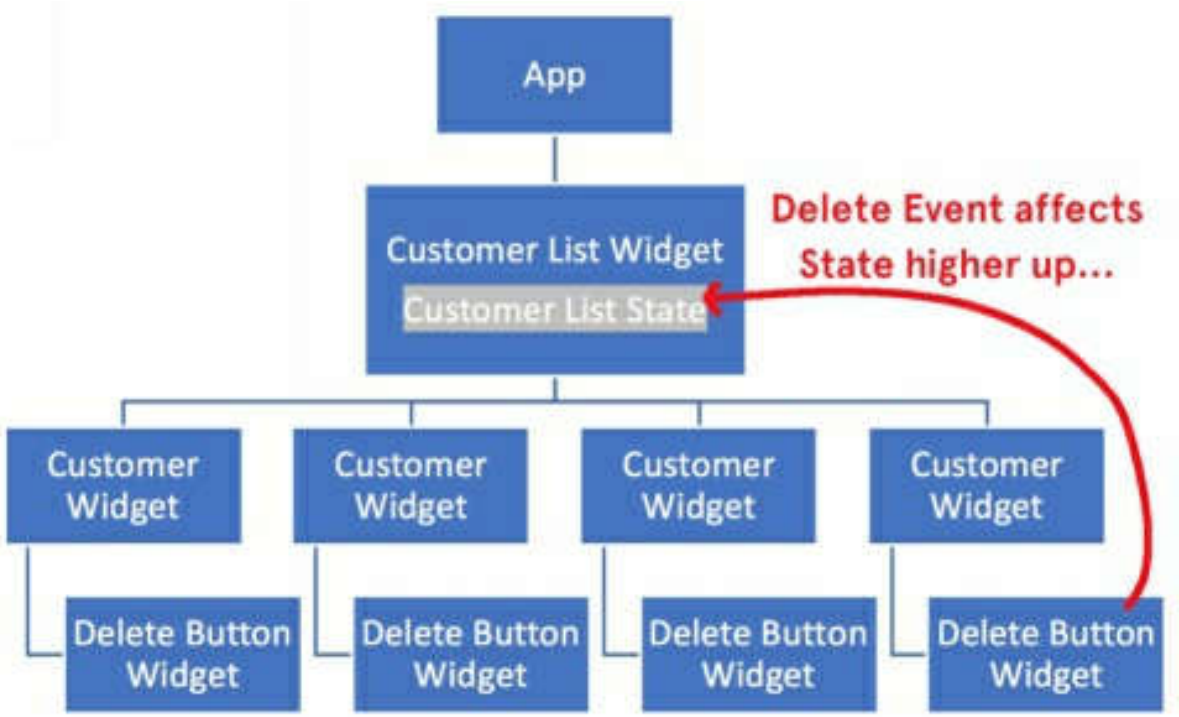
affect State at a ***similar or higher*** level up in the Object hierarchy.

For Example:

Say we have an application structured like this:

- The app displays a list of customers.
- Each customer in the list has a delete button.
- The user clicks on the delete button and the customer disappears.

If you have a list of customers in a home page and you delete the customer, then the Event may be triggered from a button in a lower-level widget but affect the customer list state, which is held in a higher-level home page widget.



State & Events – Problems

So, after reading about State and Events, we realize we have two problems:

- We need to store State in higher-level objects in the Widget tree but we need to pass that state data down to lower-level objects so it can be rendered (i.e. so its data can be put into Widgets).
    - Example.
        - Store Customer List state in higher-level Customer List Widget.
        - Pass Customer information down from Customer List Widget to Customer Widgets.

- We need to process Events and change state in higher-level objects when events occur in lower-level objects in the Widget tree.
    - Example.
        - Have Delete button events flow up from Delete Button Widgets up to Customer List Widget, affecting state in Customer List Widget.

State & Events – Different Approaches

There have been several different approaches to the problems above and we are about to cover them in more detail. Bear in mind these approaches are evolving and that there will probably be new ones by the time this book is released.

Mixing Approaches

It's all about finding out what approach you understand and like, or rolling your own. Remember you can mix these approaches. You could have an app that uses multiple InheritedWidgets, uses Streams and StreamBuilders but also uses Stateful Widgets.

How I Decide Where to Put State

When I write apps, I usually do the following:

- I put the global state (or other state shared by multiple Widgets) in one or more BLoC's.
    - I use Streams & StreamBuilders to update the UI when state changes.

- I put local state in StatefulWidgets.
    - Stateful Widgets were designed for storing local state.

State & Events – Commonly-Used Approaches

Stateful Widget Approach

- Store state in Stateful Widgets at a high-enough level in the Widget tree to ensure that the data is not repeated.
- Pass state from parent Widgets to child Widgets through the constructor.
- Pass event handler method (that modifies state) from parent Widget methods to child Widgets through the constructor. Child Widgets can then invoke method to change state in Parent Widget.

Example:

To see an example of this, see State & Stateful Widget Approach

Pros/Cons

- It works well for smaller apps.
- It doesn't work well for bigger apps.
    - It can get messy, especially if you need to pass state / event handlers though multiple levels of the Widget tree.

InheritedWidget Approach

- This approach removes most of the requirements to use Stateful Widgets, enabling the user to use Stateless Widgets instead in many cases.
- You create a 'state holder' class that acts as a Widget in the Widget hierarchy. This class extends InheritedWidget, stores the state data and has a single child widget.
- All the Widgets below this class can then be Stateless Widgets and they can use the BuildContext to access this InheritedWidget and its state data.

Example

To see an example of this, see State & InheritedWidget

Approach

Pros/Cons

- It works well for smaller apps.
- It doesn’t work well for bigger apps

Scoped Model Approach

- This approach removes most of the requirements to use Stateful Widgets, enabling the user to use Stateless Widgets instead in many cases.
- Use a 3rd party package called ScopedModel to store a state model in your Widget Tree. You can write code in your ‘build’ method of your widget and there use the Context to get a reference to this Scoped Model so that you can read and write its state.
- This works well for simple apps but is not structured enough for larger apps.

Example

To see an example of this, see State & ScopedModel Approach

Pros/Cons

- It works well for smaller apps.

BLoC w/Streams Approach

- BLoC stands for ‘Business Logic Components’.
- It’s a pattern for state management recommended by Google developers.
- It about storing the app State in a central place (a business logic object stored in a Stateful Widget) and it communicates with the rest of the app’s (mostly) Stateless Widgets using streams.

Example

To see an example of this, see Chapter State & BLoCs w/Streams Approach

Pros/Cons

- It is overkill for smaller apps.