Interaction with API

API (Application Program Interface) is the most used method for a client application such as mobile application to interact with web services or servers.

In this example, we will interact with a third-party free API and create our own simple REST API which handle database operation. Then, we will use and access these APIs via our flutter mobile app.

1. Setup the MySQL database and Apache web server

* Install either Xampp or Wamp package.
* Run the Apache Web Server service and MySQL database service.
* Create database on MySQL named **dbExpense** – Use PhpMyadmin or MySQL Workbench
* Create a table named expenses that has the following attributes:

A screenshot of a computer

Description automatically generated

* Open the Xampp/Wamp installation directory and find htdocs folder.
* Create a folder name it as api.
* Inside the created folder, create PHP file named it as expenses.php
* Install and use any file authoring tool such as Notepad++ or Visual Studio code to edit the php file.

1. Develop the REST API in expenses.php

|  |
| --- |
|  |

* The PDO (PHP Data Object) will establish connection to our database
* This php program will receive and process HTTP request with method POST or GET.
* If the request header contains Content-Type=application/json it will parse the request body data into object.
* By default, the response code is set to **404** using http\_response\_code()
* When the code execution goes into POST or GET the response code will be changed accordingly so that if the code does not go into any of the specified conditions it means that the process requested is not available hence not found (http error code 404).
* While technically it is possible for us to set any code we want but it is important to use the globally recognized standard HTTP response code <https://developer.mozilla.org/en-US/docs/Web/HTTP/Status>. Commonly used: 200 (good/success), 400 (bad request, invalid data etc), 500 (internal server error, database error etc).

1. Create a new folder, controller and create a new file “request\_controller.dart” inside it. Dart recommends using snake case for file name (<https://dart.dev/tools/linter-rules/file_names>).
2. Import the following package and define the class inside the newly created file. (install http package if not yet)

|  |
| --- |
|  |

* This Request controller class will be the boundary class handling our application interaction with APIs. By having a single class facilitating http request and response, we can omit lengthy codes when sending request. Furthermore, this approach provides better scalability and maintainability where we can simply modify this class to change how all our request works.
* The constructor in line 12 takes 2 paramater, the path is a must while server is optional parameter with default value. This **default value should be the address of your server device**. Thus, we can use this class later to communicate with 3rd party server by passing the server address while leaving out the server address to communicate with our own server.

Make sure that your mobile device and your server is in the same network and use ipconfig in your server device command line to see its IP address. *Emulated devices share same network with the computer which runs it by default.*

For **Emulated device**, you can use **10.0.0.2** as the address to refer to your computer which runs the emulator if you run your server locally.

* The method setBody() will receive the value to be sent inside Map<String,dynamic> which mimics JSON data structure in dart.
* the http.post() and http.get() is asynchronous method, thus you need to wait for its completion before processing your result which is why **await** is used to wait until the process is complete. Note that await can only be used inside **async** function/method.
* Both post() and get() method of our RequestController class is also asynchronous which returns Future<void>.

**Future** in dart is used for data that has no value **yet**. It can be used with any data type or class. Eg: Future<int>, Future<YourClass> etc.

Future data is not equivalent to actual data. (Future<int> != int) hence we can’t use it directly without converting it into normal data via **await** keyword or using it inside callback. (int == await Future<int>)

* After all request method, the \_parseResult is called which convert the string in http response into JSON format into \_resultData if possible. Dynamic is used instead of Map here due to uncertainty of the response data depending on the http status and other factors.

1. Reorganize the expense class. Create a new folder “Model” and create a new dart class expense.dart inside it. Move the previous definition of class Expense from dailyexpense.dart into the new file and improve it as shown below.

|  |
| --- |
|  |

* Import the new request controller class. It is **recommended** to use package: instead of relative path././../ etc since path might changes if you move the file around.
* The amount attribute is changed to double data type.
* The toJson() and fromJson() method is added to make our class serializable from/to json format.
* In the fromJson() method, for the amount attribute, the value from the json object in the parameter are extracted as dynamic instead of directly as double. This is important because dart will assume non decimal value as int, for example 11 will be converted to int when json\_decode is called and int can’t be assigned into double attribute directly in dart. Thus, we read it as dynamic first and convert into double to prevent such type error. *This similar issue and solution also might be applicable for other data type other than string*.
* Both save() and loadAll() method perform http request using the controller class.
* This method must be async since we need to use await within it.
* Ater awaiting the http post/get, we verify the http status using code 200 which indicate successful request.
* In the loadAll() method we uses for each loop to iterate through the result data and construct new object of expense using the named constructor “fromJson” before adding it into our list of expense which then are returned.

1. Go back to “dailyexpense.dart” and import the new files we created.



1. Now that the amount attribute of the expense class is double instead of string (previous vers). There will be error. Fixes all the error related to amount.

We no longer have to convert amount to double since it is already a double.

|  |
| --- |
| A screen shot of a computer program  Description automatically generated  Change the subtitle inside ListTile inside the \_buildListView() method to include date time. |
| A screen shot of a computer code  Description automatically generated  Change the onSave parameter inside save button of the edit expense screen. |
|  |

1. Modify the input fields to add date time input and set the initial value using 3rd party API.

|  |
| --- |
| Declare a new TextEditingController for the new date input. |
| Add new method selectDate() inside \_ExpenseListState { }  Basically, we will diplay the date picker and then time picker to get user input. |
| Alter the build() method. |
| Create a new method to display message via snackbar to simplify reusing it. |
| Create new method to calculate total from our list of expenses and display it |
| Create new/alter the init() override method.  Init() will only be called once in the widget lifecycle.  We use addPostFrameCallback to do something after the widget is initalized later   * Firstly, we request the current date time from a public 3rd party API provided by <https://worldtimeapi.org/> * We also load all the expense from server and add it into our list * Finally, we set state to update the display and call the calculateTotal method.   In this example, you can see that **await** is not the only option to **handle Future data type**. Alternatively, we can also use **then() callback** which is a bit longer in term of code but is non blocking. |

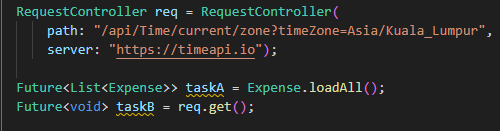
1. Alter the addExpense() method to save the data into the database

|  |
| --- |
|  |

1. Try your application. Add some data and reload the expense screen to see if it does load back the data saved into database.

Asynchronous Programming in Dart

In dart, asynchronous process involves future data type. Asynchronous function/method must always return future data type. As future data is not equivalent to normal data, we can’t use it directly. In order to process or use future data type, we can do the following approaches. *To demonstrate the difference we will be using the similar process to what has been done in the init() method of dailyexpense screen.*



Previously, we directly use the return value, now we will first store the return value into the appropriate Future variable to see what happens more clearly. We now have 2 task, taskA and taskB which both are asynchronous. TaskA will fetch expense data from our server. TaskB will fetch current time from timeapi.io. Let **assume** that **taskA takes 10 milliseconds** while **taskB takes 3 milliseconds.**

1. **await**

The first and simplest way to handle future data is using await keyword. The characteristic of await is:

* it will **block** the code execution until the awaited future completes before executing the next lines.
* It **can only be used inside async** method. “void methodname() async {}”

In most cases where only 1 asynchronous task is needed, await will most likely does the job. However, when we have 2 task which is not related to each other as shown in the example, using consecutive await is actually bad for efficiency.

A black background with text

Description automatically generated

In the code above, taskB will only be executed after taskA complete and then the code waits until taskB completes. Based on our prior assumptions this process will takes 13 milliseconds. Why should taskB wait for taskA when it is not related? *Do note that if taskB is related to taskA result then await is indeed fit to be used.*

1. **then() callback**

The then() callback is a method available to all future data type where we assign what code should be executed after the future is completed. Analogically, it is like telling a waiter to deliver food to our table when its done. After instructing the waiter we can then proceed to do anything surfing internet or whatever it is. The characteristic of then() callback is:

* It will **not block** code execution. We assign what should happen later and proceed to next lines.
* It can be used in **any** function/method regardless of async or not.

A computer code on a black background

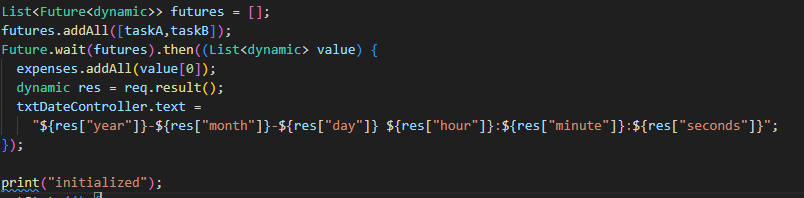
Description automatically generated

In the example above, both task is executed without waiting for it. However, we do handles it through the then callback. Based on our assumption, this code will execute at most 10 milliseconds. In reality, executing this code may have variety output. TaskB and taskA can complete faster or slower depending on external factors but the initialized will most likely be printed first.

Do note that, if taskB uses or need taskA result, then() will not be appropriate since it is uncertain which callback will be executed first since both are not waited. Thus, then() is best used for futures which is unrelated to each other. For single future it is much simpler to use await instead.

1. **grouping multiple asynchronous task**

When processing multiple futures data, using await might be inefficient and using then can produce uncertainty. Sometimes, we have code to execute only after all of the future is completed which is why grouping future data might be necessary.



Grouping future can be done by creating list of future. Since in this example we have future of variety type, dynamic is used instead. What happens, when we group multiple Futures using Future.wait() is dart will actually create a single Future which will complete when all the grouped future is done. Thus, we can use the return value of the Future.wait() same as we uses any future either using then() as shown above or using await as shown below. Index 0 is used to refer to the result of taskA since it is the first future in the list of future hence the first index(0) of the result list of dynamic will hold the value for the future at first index(0).

A computer screen with text

Description automatically generated

Both of the grouping example works but not in the same way. Similar to how individual future works, the **then()** callback version is non blocking thus, you will see initialized printed first. While the await **version** will block the execution making the initialized printed only after both tasks completed.

The previous examples are extended with declarations to clarify and make it easier to understand. The overall code can be shortened as follows.

A screen shot of a computer code

Description automatically generated

Generally, await is more preferred due to its simplicity and future grouping solve the issue of unnecessary awaiting independent tasks.