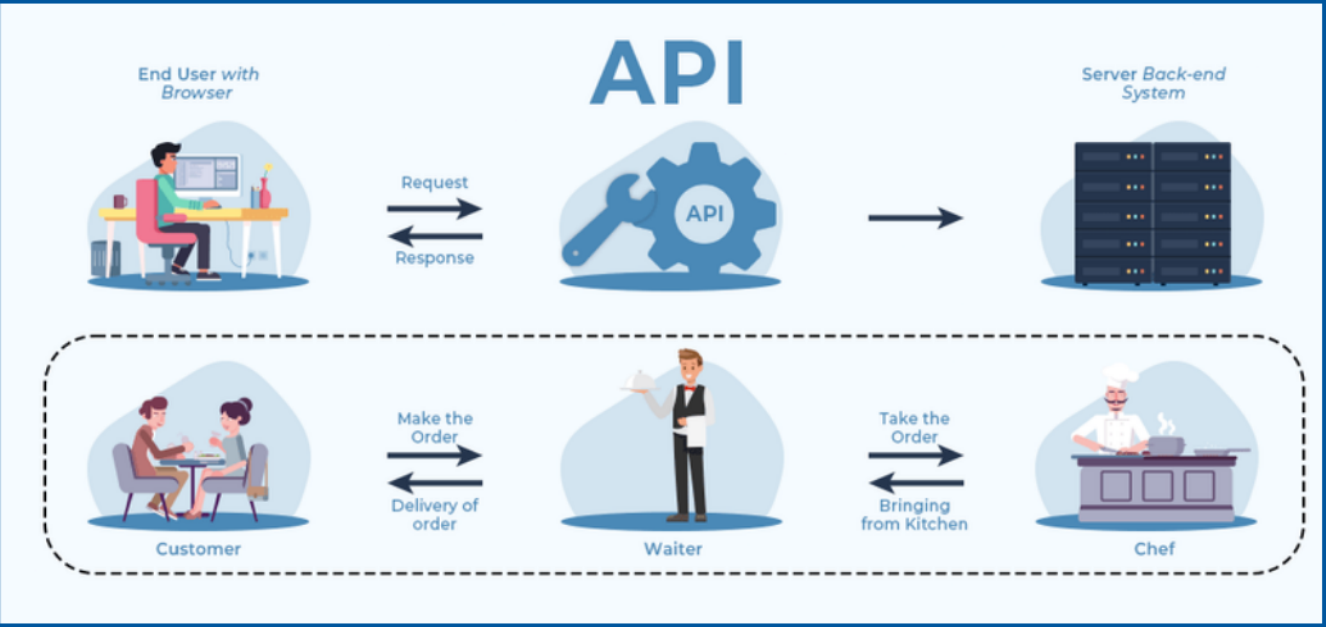
Flutter cheat Sheet

Creating Dynamic App

- **API Introduction :-**

Integrating APIs (Application Programming Interfaces) in Flutter allows your app to communicate with external services, retrieve data, and perform various operations. Here's an introductory guide to working with APIs in Flutter:



A developer extensively uses APIs in his software to implement various features by using an API call without writing complex codes for the same. We can create an API for an operating system, database system, hardware system, JavaScript file, or similar object-oriented files. Also, an API is similar to a GUI(Graphical User Interface) with one major difference. Unlike GUIs, an application program interface helps software developers to access web tools while a GUI helps to make a program easier to understand for users.

Flutter cheat Sheet

- **API Types :-**

1. Get Api
2. Post Api
3. Put Api
4. Patch Api
5. Delete Api

- **Get Api :-**

In the context of web development and APIs, a "GET API" refers to an API endpoint that is designed to retrieve data from a server. It's one of the most common types of API endpoints, especially in RESTful API architectures.

```
import 'package:http/http.dart' as http;

void fetchData() async {
  var url = Uri.parse('https://api.example.com/data');
  var response = await http.get(url);

  if (response.statusCode == 200) {
    // Successful GET request
    print('Response body: ${response.body}');
  } else {
    // Handle error
    print('Error: ${response.reasonPhrase}');
  }
}
```

Flutter cheat Sheet

- **Post Api :-**

A "POST API" refers to an API endpoint that is designed to accept data from clients and create or update resources on the server. Unlike a GET API, which is used for retrieving data, a POST API is used for submitting data to the server.
When you make a POST request to a POST API endpoint, you typically include a payload, which contains the data that you want to send to the server. This data can be in various formats such as JSON, XML, or form data.

```
void postData() async {
  var url = Uri.parse('https://api.example.com/data');

  Map<String, String> headers = {
   'Content-Type': 'application/json',   };

  Map<String, dynamic> requestBody = {
   'key1': 'value1',  'key2': 'value2',   };

  String jsonBody = json.encode(requestBody);

  var response = await http.post(  url,
   headers: headers,  body: jsonBody,   );

  if (response.statusCode == 201) {
   print('Response body: ${response.body}');
  } else {
   print('Error: ${response.reasonPhrase}');   } } }
```

Flutter cheat Sheet

- **Put Api :-**

A "PUT API" refers to an API endpoint that is designed to update existing resources on the server with the data provided in the request. Similar to POST requests, PUT requests are used to modify data on the server. However, PUT requests are typically used when you want to update an entire resource with new data, rather than appending or modifying specific parts of it.

```
void putData() async {
var url = Uri.parse('https://api.example.com/data/123');

  Map<String, String> headers = {
   'Content-Type': 'application/json',   };

  Map<String, dynamic> requestBody = {
   'key1': 'updatedValue1',   'key2': 'updatedValue2',   };

  String jsonBody = json.encode(requestBody);

  var response = await http.put(  url,  headers: headers,
   body: jsonBody,   );

  if (response.statusCode == 200) {
   print('Response body: ${response.body}');  } else {
   print('Error: ${response.reasonPhrase}');   }  }  }
```

Flutter cheat Sheet

- **Patch Api :-**

A "PATCH API" refers to an API endpoint that is designed to perform partial updates to existing resources on the server. Unlike PUT requests, which are used to replace an entire resource with new data, PATCH requests are used to update specific parts of a resource while leaving the rest of it unchanged.

```
void patchData() async {
  var url = Uri.parse('https://api.example.com/data/123');

  Map<String, String> headers = {
   'Content-Type': 'application/json', };

  Map<String, dynamic> patchData = {
   'key1': 'updatedValue1', 'key2': 'updatedValue2',  };

  String jsonPatchData = json.encode(patchData);

  var response = await http.patch(   url,
   headers: headers,  body: jsonPatchData,   );

  if (response.statusCode == 200) {
   print('Response body: ${response.body}');
  } else { print('Error: ${response.reasonPhrase}'); } }
```

Flutter cheat Sheet

- **Delete Api :-**

A "DELETE API" refers to an API endpoint that is designed to remove or delete a resource from the server. When you make a DELETE request to a DELETE API endpoint, you are essentially asking the server to delete the specified resource.

```
void deleteData() async {
  var url = Uri.parse('https://api.example.com/data/123');

  // Make the DELETE request
  var response = await http.delete(url);

  // Check the response status code
  if (response.statusCode == 200) {
    // Successful DELETE request
    print('Resource deleted successfully');
  } else {
    // Handle error
    print('Error: ${response.reasonPhrase}');
  }
 }
}
```

Flutter cheat Sheet

- **Api Exception Handling :-**

Handling API exceptions in Flutter involves catching and properly managing errors that may occur during API requests, such as network errors, server errors, or client-side errors. The http package provides mechanisms for handling such exceptions.

```
void fetchData() async {
  try {
    var url = Uri.parse('https://api.example.com/data');
    var response = await http.get(url);
    if (response.statusCode == 200) {
      print('Data: ${response.body}');
    } else {
      print('Server Error: ${response.reasonPhrase}');   }
  } on http.ClientException catch (e) {
    print('Client Error: $e');
  } on http.ServerException catch (e) {
    print('Server Error: $e');
  } on FormatException catch (e) {
    print('Format Error: $e');
  } catch (e) {    print('Error: $e');  }   }
```