Mastering Dio Package for HTTP Network API Calls in Flutter

Rahul Agarwal

Flutter Developer & Trainer

in in/rahul-connect

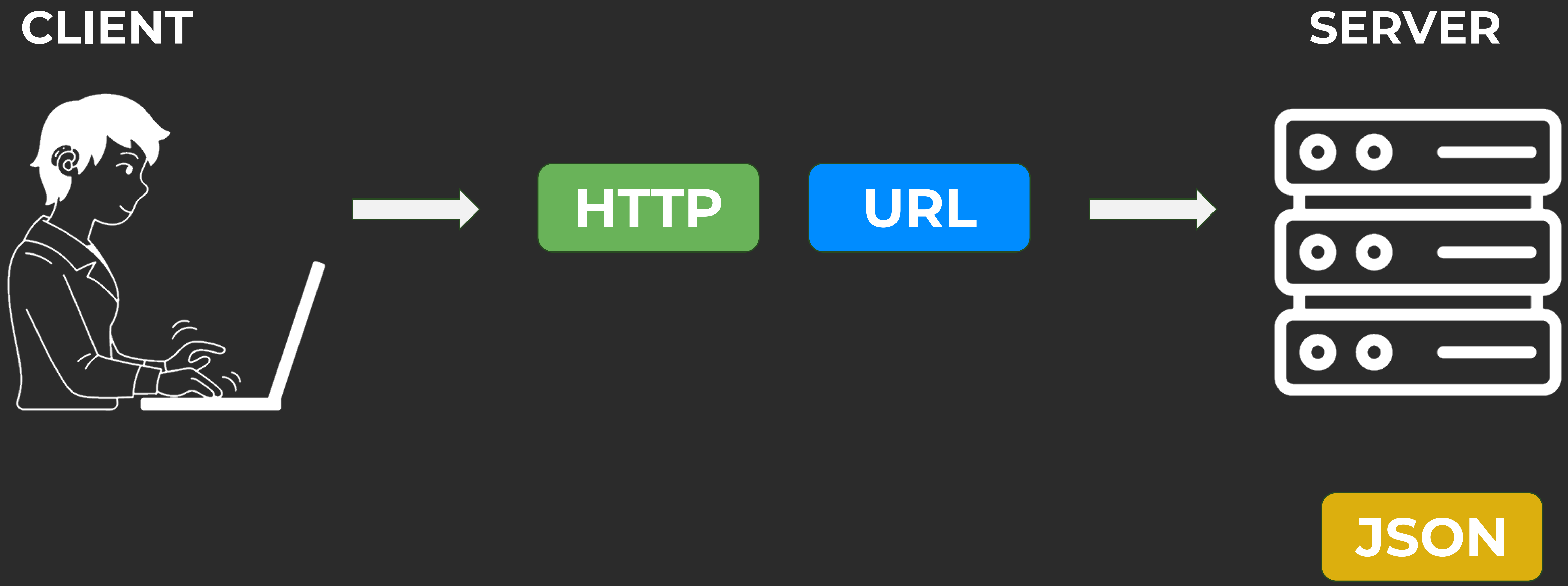
Course Overview

- Understand the fundamentals of APIs and their functionality
- Justify the use of the Dio library for efficient HTTP communication
- Explore the provided Starter Project as a foundation for practical learning.
- Execute various HTTP requests such as GET, POST, PUT and DELETE
- Implement Dart models to streamline JSON Serialization and Deserialization processes.
- Master Error Handling and status code interpretation for robust application behavior.
- Utilize Dio Interceptors to customize both requests and responses effectively.
- Implement Bearer and Basic Authentication mechanisms for secure access to private routes.
- Integrate Shared Preferences for storing and managing authentication tokens.
- Explore Dio's capabilities in handling File Uploads and Downloads.
- Implement reliable and persistent Caching using Dio Cache Interceptor and Hive Store.
- Discuss advanced tips and strategies for Optimizing network performance with Dio.

droidcon academy

Getting Started

What is an API ?



Droidcon Cafe

Concept of Asynchronous Programming and Futures ?

droidcon academy

Droidcon Cafe

Token 2
Token 1
Token 1 Ready
Kitchen

Why use Dio ?

- Dio is a powerful and easy-to-use package in Flutter designed for making HTTP requests.
- It simplifies the process of integrating network functionalities into Flutter applications.
- Dio provides robust support for asynchronous operations.
- Easily manipulate requests and responses using Dio's interceptor capabilities.
- Dio simplifies handling form data and file uploads effortlessly.
- Dio benefits from a strong community and active maintenance.
- Continuous improvement ensures compatibility with the latest Flutter releases.
- The adoption of Dio emerges as a strategic and professional choice for Flutter developers.

Implement GET request and JSON data parsing

droidcon academy

Network CRUD Operations

CRUD operations are fundamental for APIs as they perform a standardized set of actions for manipulating data. It stands for Create, Read, Update, and Delete.

These operations perform the following requests :

- **GET** : This basically helps you request data from the server. The request may include some query parameters to specify the type of data that you want to retrieve.
- **POST** : This is used to send data to a server to create a new resource. The data sent to the server with POST is stored in the body parameter of the request.
- **PUT** : It updates existing data on the server. PUT is used to replace the entire resource with a new representation, meaning that all the fields of the resource are sent in the request body, even if they are not modified
- **DELETE** : As the name implies, it deletes your requested data from the server.

Use of Data Models

- A data model represents the structure and attributes of a specific type of data in an application.
- It defines the properties, methods, and relationships associated with the data object.
- In Dart, data models are typically implemented using **classes** which help us to determine the structure of the data.

```json
{
    "id":1,
    "name":"Rahul"
}
```

```dart
class User {
  final int id;
  final String name;
  User(this.id,this.name);

 factory User.fromJson(Map jsonData){
    return User(jsonData['id'], jsonData['name']);
  }
}
```

A Guided Code Challenge with Solution

Implement POST, PUT and DELETE request

Handling Exceptions and Interpreting Status Codes

Why Handling Exceptions is Important ?

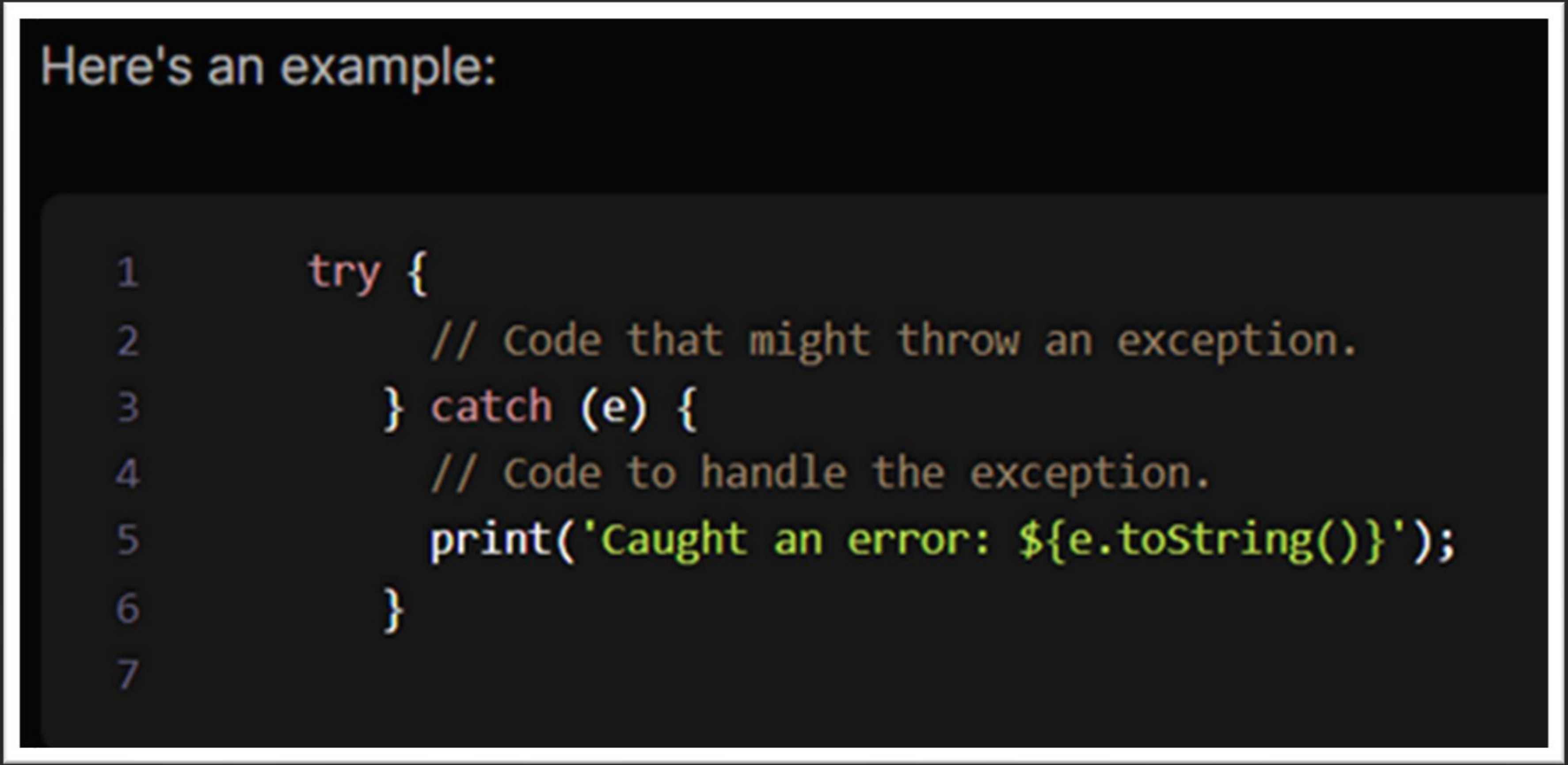
Error vs Exception

- An **Error** in Dart refers to a severe issue that leads to the termination of the Dart program. These are critical issues that we cannot catch and handle. Errors include but not limited to OutOfMemoryError, StackOverflowError, and AssertionError.

- An **Exception** in Dart is something unexpected that occurs during the execution of a program. These exceptions can be caught and handled, thus ensuring the program does not terminate abruptly. Examples include FileNotFoundException, SocketException, and FormatException. Therefore, effectively handling exceptions in Flutter is integral to ensure the program does not crash during the execution.

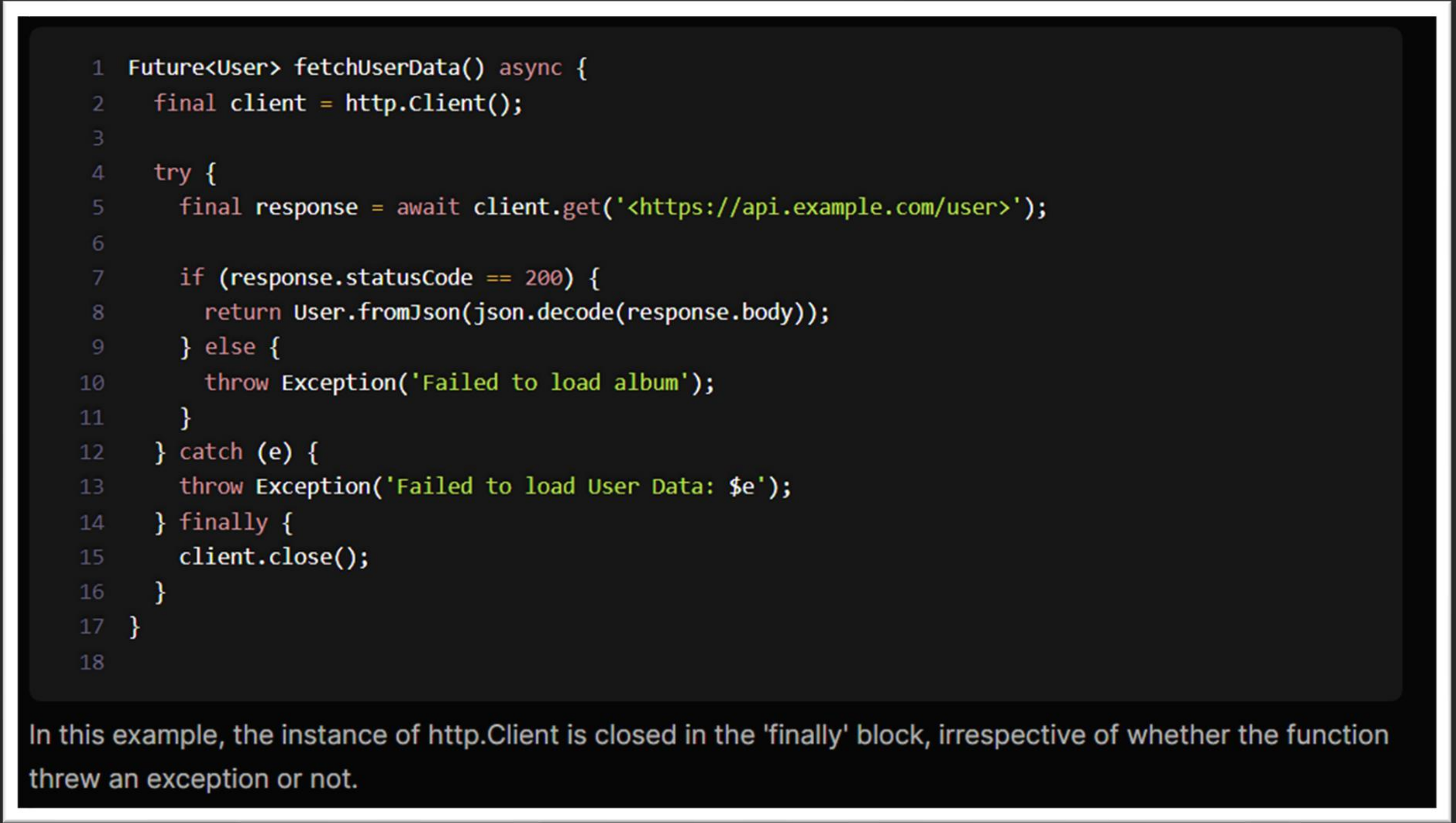
Understanding Try-Catch Blocks

- The **‘ try ’** block contains code that could potentially throw an error.
- If an exception occurs, control is passed to the **‘ catch ’** block, which handles the error.
- A **‘ Finally ’** block can be appended after try catch which will always execute regardless of whether an error occurred or not.



Here's an example:

```
try {
    // Code that might throw an exception.
  } catch (e) {
    // Code to handle the exception.
    print('Caught an error: ${e.toString()}');
  }
```



```
Future<User> fetchUserData() async {
  final client = http.Client();

  try {
    final response = await client.get('<https://api.example.com/user>');

    if (response.statusCode == 200) {
      return User.fromJson(json.decode(response.body));
    } else {
      throw Exception('Failed to load album');
    }
  } catch (e) {
    throw Exception('Failed to load User Data: $e');
  } finally {
    client.close();
  }
}
```

In this example, the instance of http.Client is closed in the 'finally' block, irrespective of whether the function threw an exception or not.

DioExceptions Class

- Since we are using Dio package, it will wrap the error or exception class of Dart into **DioException** and give us that. So while handling exceptions we will use **on DioException** with the try catch block.

When an error occurs, Dio will wrap the Error/Exception to a DioException.

```dart
  try {
  // 404
  await dio.get('https://api.pub.dev/not-exist');
} on DioException catch (e) {
  // The request was made and the server responded with a status code
  // that falls out of the range of 2xx and is also not 304.
  if (e.response != null) {
    print(e.response.data)
    print(e.response.headers)
    print(e.response.requestOptions)
  } else {
    // Something happened in setting up or sending the request that
triggered an Error
    print(e.requestOptions)
    print(e.message)
  }
}
```

DioException Object will contain :

```dart
// The request info for the request that throws exception.
RequestOptions requestOptions;

// Response info, it may be `null` if the request can't reach to the
// HTTP server, for example, occurring a DNS error, network is not available
Response? response;

// The type of the current [DioException].
DioExceptionType type;

// The original error/exception object;
// It's usually not null when `type` is [DioExceptionType.unknown].
Object? error;

// The stacktrace of the original error/exception object;
// It's usually not null when `type` is [DioExceptionType.unknown].
StackTrace? stackTrace;

// The error message that throws a [DioException].
String? message;
```

droidcon academy

What are Status Codes ?

Status codes are three-digit numbers that are returned in the HTTP response from a server to indicate the status of a request made by a client. They provide information about whether the request was successful, encountered an error, or requires further action.

They are grouped into the following :

- **Informational** (1xx): These status codes means that the server has received the request and is continuing the process.
- **Success** (2xx): These status codes indicate that the request was successfully received, understood, and processed by the server.
- **Redirection** (3xx): These status codes indicate that further action is needed to fulfill the request. They often indicate that the client needs to redirect to a different URL.
- **Client Errors** (4xx): These status codes indicate that the client has made an error in the request. It could be due to invalid parameters, authentication issues, or unauthorized access.
- **Server Errors** (5xx): These status codes indicate that the server encountered an error while processing the request. It could be due to server overload, internal server errors or unavailability.

Throwing Exceptions Based on Status Codes

```
Future<void> deletePost({required int id}) async {
  try {
     await _dio.delete('/posts/$id/fake’);

  } on DioException catch(e){

    if (e.response?.statusCode == 404) {
      // Not Found
      throw Exception('Resource not found');
    } else if (e.response?.statusCode == 401) {
      // Unauthorized
      throw Exception('Unauthorized access');
    } else if (e.response?.statusCode == 500) {
      // Internal Server Error
      throw Exception('Internal server error');
    } else {
      // Other status codes
      throw Exception('Unexpected Error. Failed to delete data.');
    }
  }
}
```

droidcon academy

Implementing Interceptor and building Authentication UI

What are Interceptors ?

Interceptors are like a middleware which means they acts a the layer between the client and server that modifies the requests and responses intercepting the HTTP request. They are particularly useful for scenarios where you need to apply the same logic across multiple requests or responses.

Some of its use cases are :

- Modify request parameters, headers, or authentication tokens before sending the request to the server.
- Modify or log the response data, headers, or status codes after receiving the response.
- Handle errors globally, such as adding a default error behavior, logging error messages and directly map them to Error classes that our app can handle.
- Implement caching strategies by intercepting requests and responses to store or retrieve data from a cache.
- Retry failed requests automatically by intercepting errors and making subsequent attempts.
- Transform data before it is sent or after it is received, such as converting JSON responses to Dart objects.

Interceptor Methods

Interceptors will help us implement all the discussed use cases by giving us specific callbacks. We have onRequest, onResponse and onError methods available in the Interceptor class, and we can override them as well.

- onRequest Method : Modify requests before they are sent. For example, you might want to add authentication headers, logging, or any other custom headers.
- onResponse Method : Modify responses after they are received. You can modify data, log information, or perform any other post-processing.
- onError Method : Handle errors that occur during the request. For example, you can globally handle authentication errors or log specific error messages

```
dio.interceptors.add(
 InterceptorsWrapper(
  onRequest: (RequestOptions options, RequestInterceptorHandler handler) {
   // Do something before request is sent.
   // If you want to resolve the request with custom data,
   // you can resolve a `Response` using `handler.resolve(response)`.
   // If you want to reject the request with a error message,
   // you can reject with a `DioException` using `handler.reject(dioError)`.
    return handler.next(options);
  },
  onResponse: (Response response, ResponseInterceptorHandler handler) {
   // Do something with response data.
   // If you want to reject the request with a error message,
   // you can reject a `DioException` object using `handler.reject(dioError)`.
    return handler.next(response);
  },
  onError: (DioException error, ErrorInterceptorHandler handler) {
   // Do something with response error.
   // If you want to resolve the request with some custom data,
   // you can resolve a `Response` object using `handler.resolve(response)`.
    return handler.next(error);
  },
 ),
);
```

Interfere with Execution Flow

In all interceptors, we can interfere with their execution flow with the help of handler object. The available methods on the handler are :

- **Resolve** : The resolve callback returns a response directly without executing other interceptors.
- **Reject** : The reject callback completes the request without executing other interceptors.
- **Next** : The next callback allows to continues the flow and call the next interceptors.

Note : If you return null in onRequest, the request will be canceled, and no further processing will occur.

```
dio.interceptors.add(
 InterceptorsWrapper(
  onRequest: (RequestOptions options, RequestInterceptorHandler handler) {
  // Do something before request is sent.
  // If you want to resolve the request with custom data,
  // you can resolve a `Response` using `handler.resolve(response)`.
  // If you want to reject the request with a error message,
  // you can reject with a `DioException` using `handler.reject(dioError)`.
    return handler.next(options);
  },
  onResponse: (Response response, ResponseInterceptorHandler handler) {
  // Do something with response data.
  // If you want to reject the request with a error message,
  // you can reject a `DioException` object using `handler.reject(dioError)`.
   return handler.next(response);
  },
  onError: (DioException error, ErrorInterceptorHandler handler) {
  // Do something with response error.
  // If you want to resolve the request with some custom data,
  // you can resolve a `Response` object using `handler.resolve(response)`.
   return handler.next(error);
  },
 ),
);
```

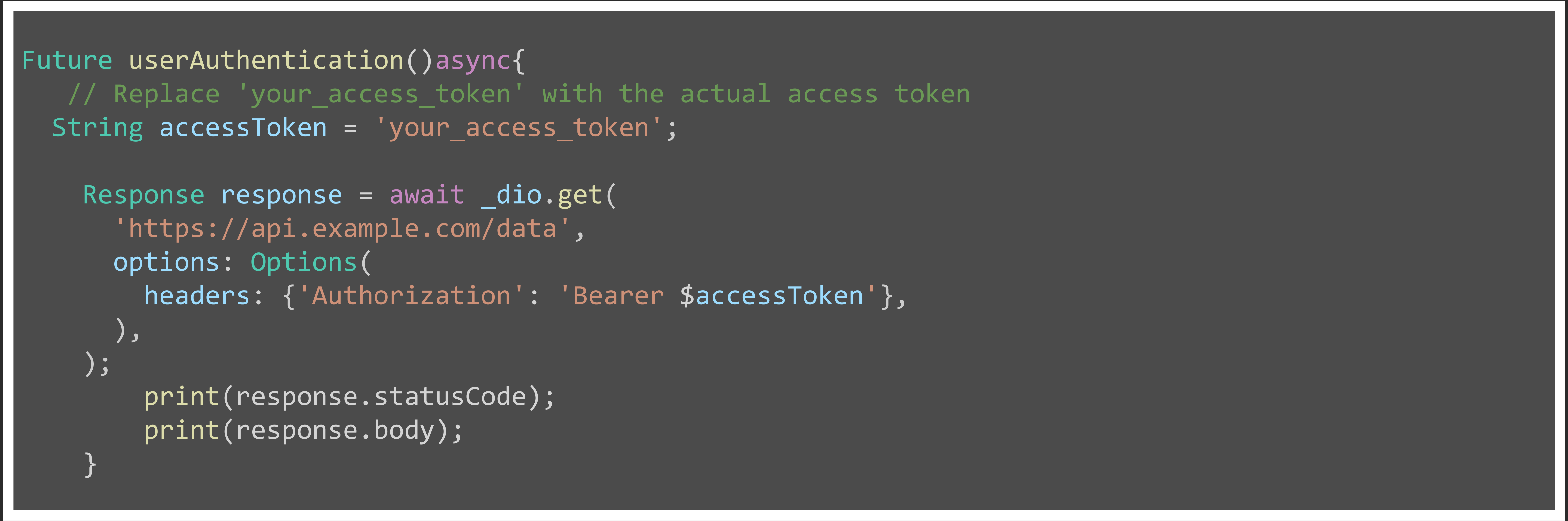
Authentication State Handling with Shared Preferences

Unlock Secure Routes with Bearer & Basic Authentication

Make Authenticated requests

Most of the time when building application, in order to fetch or access data from any web services, you need to provide authorization. There are two ways to do this :

1. **Bearer Authentication** : Bearer authentication is a type of token-based authentication where a bearer token, usually a JSON Web Token (JWT) or an OAuth token, is included in the request header. We have to use Bearer as prefix along with token value.



```
Future userAuthentication()async{
   // Replace 'your_access_token' with the actual access token
  String accessToken = 'your_access_token';

    Response response = await _dio.get(
      'https://api.example.com/data',
      options: Options(
        headers: {'Authorization': 'Bearer $accessToken'},
      ),
    );
        print(response.statusCode);
        print(response.body);
    }
```

Make Authenticated requests

2. Basic Authentication : Basic authentication means that the client app sends the username and password to the server in the request header, encoded in Base64. It is less secure than bearer tokens and often used in conjunction with HTTPS. Also we have to use Basic as prefix along with value.

```
Future userAuthentication(String username,String password) async {

    String basicAuth = 'Basic ${base64Encode(utf8.encode('$username:$password'))}';

    Response response = await _dio.get(
      'https://api.example.com/data',
      options: Options(
        headers: {'Authorization': basicAuth},
      ),
    );
        print(response.statusCode);
        print(response.data);
    }
```

Request Cancellation

- Cancellation in Dio refers to the ability to cancel an ongoing HTTP request that has been sent using the Dio instance.
- A CancelToken is created using the CancelToken() constructor, and can then be passed as an optional parameter to the Dio methods that send HTTP requests.

```
final cancelToken = CancelToken();

_dio.get("https://api.example.com/data", cancelToken:cancelToken).catchError((DioException error) {
  if (CancelToken.isCancel(error)) {
    print('Request canceled: ${error.message}');
  } else {
    // handle error.
  }
});

// Cancel the requests with "cancelled" message.
cancelToken.cancel('cancelled');
```

Handling Timeout

- Timeouts refer to the maximum duration a network request is allowed to take before it is considered as failed.
- You can handle timeouts using the connectTimeout and receiveTimeout properties of the Options class.

```
final Dio _dio = Dio(BaseOptions(
    baseUrl: " https://api.example.com/",
    contentType: 'application/json; charset=utf-8’,
    responseType: ResponseType.json,
    connectTimeout: const Duration(seconds: 10),
    receiveTimeout: const Duration(seconds: 10),
  ))
```

Handling File Uploads and Downloads

Dio Caching and Network Optimization Strategies

What is Caching ?

Caching is one of the most important features to have in your application. Caching is the process of storing the response from a request on device storage to avoid frequent network requests to the API or any other resource on the internet.

Why it is Important ?

- Caching allows frequently accessed data or resources to be stored closer to the user or application, reducing the time it takes to retrieve them. This leads to faster response times and improved user experience.
- Caching helps conserve network bandwidth by reducing the need to repeatedly download the same data. Instead of fetching the same resources over the network, cached copies can be used locally.
- Caching is a key strategy for improving the scalability of systems. By reducing the load on backend servers or databases, caching helps distribute the load more evenly, allowing systems to handle more users or requests.
- Caching can result in cost savings by reducing the load on servers and optimizing resource utilization. It can also contribute to lower operational costs associated with data storage and retrieval.
- Cached data can be used to provide offline access to certain resources or functionalities. This is especially useful in mobile applications or scenarios where network connectivity is intermittent or unavailable.

Strategies for Optimizing Performance

- **Use Connection Pools :** Dio uses an HttpClient under the hood, which maintains a connection pool. Make sure to reuse the Dio instance across your application to benefit from connection pooling and reduce the overhead of establishing new connections for each request.
- **Compression :** Enable compression for responses if the server supports it. This can significantly reduce the amount of data transferred over the network.
- **Caching :** Leverage the built-in caching capabilities provided by Dio or any external packages, particularly when dealing with frequently requested and static data. To enable compression with Dio, you can set the 'Accept-Encoding' header in your requests to inform the server that your client supports compression.educe the amount of data transferred over the network.
- **Use Interceptors Wisely :** While interceptors can be powerful, using too many or unnecessary interceptors can impact performance. If you have multiple interceptors, be mindful of their execution order. Order them in a way that optimizes performance and ensures that the most critical interceptors are executed first.

Strategies for Optimizing Performance

- **Timeouts :** Set reasonable timeouts for requests to prevent waiting indefinitely. Use the Options class to configure timeout values for specific requests.
- **Batch Requests :** Combine multiple requests into a single batch when possible. This reduces the overhead of making multiple network calls and can improve overall efficiency.
- **Background Execution :** For long-running requests, consider executing them in the background using plugins like flutter_isolate to prevent UI freezes.
- **Testing and Profiling :** Regularly test and profile your Dio requests, especially in scenarios where performance is critical. Tools like Dart DevTools can help identify potential bottlenecks.
- **Update Dependencies :** Keep your Dio and related packages up-to-date to benefit from performance improvements, bug fixes, and new features.

Summary

- Understood the fundamentals of APIs and their functionality
- Justified the use of the Dio library for efficient HTTP communication
- Explored the provided Starter Project as a foundation for practical learning.
- Executed various HTTP requests such as GET, POST, PUT and DELETE
- Implemented Dart models to streamline JSON Serialization and Deserialization processes.
- Mastered Error Handling and status code interpretation for robust application behavior.
- Utilized Dio Interceptors to customize both requests and responses effectively.
- Implemented Bearer and Basic Authentication mechanisms for access to private routes.
- Integrated Shared Preferences for storing and managing authentication tokens.
- Explored Dio's capabilities in handling File Uploads and Downloads.
- Implemented reliable and persistent Caching using Dio Cache Interceptor and Hive Store.
- Discussed advanced tips and strategies for Optimizing network performance with Dio.

droidcon academy

Thank You!

Rahul Agarwal

Flutter Developer & Trainer

in in/rahul-connect