SDK Adoption Guidelines

1. Introduction

An SDK is created to facilitate and streamline the development process for integrating software applications with a specific platform, service, or API. It equips developers with ready-to-use tools, libraries, documentation, and sample codes, allowing them to efficiently utilise the features and functionality of the platform or service without having to build everything from scratch. By providing a standardized and simplified development framework, an SDK accelerates the implementation, reduces development time, and enhances the overall developer experience.

This guide delves into various stages of developing an SDK for mobile money APIs. The content presented here draws upon our extensive experience creating SDKs using various technologies such as PHP, Java, JavaScript, Node.js, and Android Java for Mobile Money APIs. We have explained how to adapt the MMAPI SDK to create a new SDK for a different Mobile Money API.

2. Environment Setup

2.1. Which programming language use?

Any programming language can be used to create an SDK, and the choice is dependent on the target users. In this document, we are using Java as a model language to explain the steps to create an SDK for Mobile Money APIs. We recommend you start with the most useful programming language for your technology developers and increase the number of languages when needed.

2.2. Selecting the right platform

Any GIT repository and versioning system can be used to manage your SDKs. We used GitHub, a widely adopted platform for managing private and public source code, to prepare this guide. To begin, you must initiate the process by creating a private repository. After completing the development and testing phases, you can make your repository public to publish your code for other developers to access and utilise.

After creating your repository, your repository URL looks something like this.

https://github.com/{githubaccount_name}/{repository_name}

Within a repository, you can manage the code across multiple branches, allowing you to organize and maintain separate versions such as in-development code, code for testing purposes, and code ready for publication.

During the creation of the GSMA MMAPI SDKs, we chose the approach shared below, we recommend you use a similar approach.

- **Staging Branch:** The Staging branch serves as the primary development environment where code is placed during the development process. Here, unit tests and integration tests are conducted. Once all the tests pass successfully, the code is then transitioned to the development branch.
- **Develop Branch:** The Develop branch is where the code is held for verification and code review. It serves as a dedicated branch for collaborative review and code refinement before further progression.
- **Main Branch:** The Main branch consists of the ultimate deliverable code for the customer, following thorough testing and verification. This branch contains the stable and finalized version of the code, ready to be published.

2.3. Version Control

When a bug or feature is reported in the existing code, modifications need to be made and verified by running tests. A feature branch is created from the staging branch, named as **feature/{name-of-the-bug}** to facilitate this process. The necessary code changes are implemented on this feature branch, and once the tests run successfully, the branch is merged back into the staging branch.

2.4. File name conventions

The common Java file naming conventions are followed here. Packages contain lowercase with dot separation. Classes use upper camel case, while variables and methods use lower camel case.

The names of all the request classes were decided based on the use case name, for example, **AccountLinkingRequest** is related to the use MMAPI use case **Account Linking**. Each API request was decided based on the API endpoints itself and rephrased to be meaningful method names. For example, the **viewRequestState** method was named after the endpoint **/requeststates/{serverCorrelationId}**. We prefix the method name with **view** because the request is a **GET**. Below is the full construction of the method **viewRequestState**. Similarly, we use the prefix **create** when the request is a **POST**.

```
1 public AsyncResponse viewRequestState(final String serverCorrelationId) throws MobileMoneyException {
2    String resourcePath = API.CREATE_REQUEST_STATES.replace(Constants.SERVER_CORRELATION_ID, serverCorrelationId);
3    return createRequest(HttpMethod.GET, resourcePath, null, null, null, AsyncResponse.class);
4 }
```

3. Comparing your API to the GSMA MM API Spec

3.1. Selecting Use Cases and User Scenarios

The GSMA Mobile Money API provides API Specifications for common mobile money use cases. The MMAPIs you choose when developing an SDK can be compared to the use cases described below, and based on how similar they are, functions can be reused.

The developer documentation for the GSMA MMAPI can be found at https://developer.mobilemoneyapi.io/api-versions-1.2/, where you can get a comprehensive understanding of each use case. The complete list of MMAPI use cases is provided below.

1. Merchant payments
2. Disbursements
3. International transfers
4. P2P transfers
5. Recurring payments
6. Account linking
7. Bill payments
8. Agent Services

Based on the following criteria, you can compare the use cases and highlight any similarities. Then you may compare the current use case code snippets and indicate the necessary adjustments.

1. API URL
    - The API URL can sometimes carry the parameters, hence when declaring the URL these parameters must be appended.
    - In MMAPI, a function **getResourcePath** is declared where the parameters must be passed and the function returns the final URL.

```
protected static String getResourcePath(final String requestEndPoint, Identifiers identifiers)
    throws MobileMoneyException {
    if (identifiers == null || identifiers.getIdentifiers() == null || identifiers.getIdentifiers().isEmpty()) {
        throw new MobileMoneyException(
            new HttpErrorResponse.HttpErrorResponseBuilder(Constants.INTERNAL_ERROR_CATEGORY,
                Constants.GENERIC_ERROR_CODE).errorDescription(Constants.NULL_VALUE_ERROR).build());
    }

    String resourcePath;
     if (identifiers.getIdentifiers().size() == 1) {
        resourcePath = requestEndPoint
            .replace(Constants.IDENTIFIER_TYPE, identifiers.getIdentifiers().get(0).getKey())
            .replace(Constants.IDENTIFIER, identifiers.getIdentifiers().get(0).getValue());
     } else {
        resourcePath = requestEndPoint.replace(Constants.MULTI_IDENTIFIER,
            identifiers.getIdentifiers().stream()
                .map(identifier -> identifier.getKey().concat("@").concat(identifier.getValue()))
                .collect(Collectors.joining("$")));
```

```
20    }
21
22    if (StringUtils.isNullOrEmpty(resourcePath)) {
23        throw new MobileMoneyException(
24                new HttpErrorResponse.HttpErrorResponseBuilder(Constants.INTERNAL_ERROR_CATEGORY,
25                        Constants.GENERIC_ERROR_CODE).errorDescription(Constants.NULL_VALUE_ERROR).build());
26    }
27    return resourcePath;
28 }
```

2. Headers of the API
    - A set of headers would be common for all the requests. These headers can be declared as global values. These would be declared in the authentication class (**MobileMoneyAuthentication**) as they would not be modified frequently.
    - Some headers would be request-specific. In that case, it's recommended to append the header values inside that specific API method to the existing set of headers. Once the request is completed and if the header values are no longer required, they can be reset to default values. This helps to prevent future API requests from passing the same header.

```
1  protected void setAuthHeaders() throws Exception {
2     switch (SecurityLevel.valueOf(this.configurationMap.get(Constants.SECURITY_LEVEL))) {
3        case NONE: break;
4        case DEVELOPMENT: {
5           this.headers.put(Constants.API_KEY, this.apiKey);
6           this.headers.put(Constants.AUTHORIZATION_HEADER, Constants.BASIC + generateBase64String());
7           break;
8        }
9        case STANDARD: {
10          this.headers.put(Constants.API_KEY, this.apiKey);
11          this.headers.put(Constants.AUTHORIZATION_HEADER, Constants.BEARER + getAccessToken());
12          break;
13       }
14       case ENHANCED: break;
15       default: throw new Exception("Undefined security level: " + SecurityLevel.values());
16    }
17 }
```

3. Request and Response body format
    - The request body is passed as **JSON string** in MMAPI call. This would be declared as a class and an object would be created with the required fields. For Example, in Java we create them as Model classes and create an object with the required fields. This object will be passed along when we make the final API call.
    - Based on the type of data returned from an API request, we need to make the appropriate Model class for each type of return object. In the case of MMAPI, the return object is **JSON string** and is transformed into the respective Model class object.

3.2. Mapping the API and Flows

Below, we evaluate two use cases that make use of several Mobile Money APIs to demonstrate how to adjust the code accordingly.

1. Individual Disbursement
2. Obtain a Disbursement Organisation Balance

Use Case 1: Individual Disbursement

The Disbursement use case of the Mobile Money APIs allows an organization to disburse funds to mobile money recipients. In the **individual disbursement** scenario, a **POST** request is made to the API endpoint **/transactions/type/disbursement** with a JSON as the request body. The response notification is sent to the provided callback URL.

	MMAPI	Another API	Similar
API URL	/transactions/type/disbursement	/disbursement/v1_0/deposit	No
Headers	Authorization, X-Callback-URL, Content-Type, X-CorrelationID and X-API-Key	Authorization, X-Callback-Url, Content-Type, X-Reference-Id, Ocp-Apim-Subscription-Key and X-Target-Environment	No
Request type	POST	POST	Yes

Request body	<pre>{
	"amount": "200.00",
	"debitParty": [
	{
	"key": "accountid",
	"value": "2999"
	}
	],
	"creditParty": [
	{
	"key": "accountid",
	"value": "2999"
	}
	],
	"currency": "RWF"
	}</pre>
		<pre>{
		"amount": "string",
		"currency": "string",
		"externalId": "string",
		"payee": {
		"partyIdType": "MSISDN",
		"partyId": "string"
		},
		"payerMessage": "string",
		"payeeNote": "string"
		}</pre>
			No

Response body	<pre>{
	"payLoad": "{\"serverCorrelationId\":\"c60db955-1e21-4d42-b7b1-385e65858383\",\"status\":\"pending\",\"notificationMethod\":\"callback\",\"objectReference\":\"33501\",\"pollLimit\":100}",
	"success": true,
	"responseCode": "ACCEPTED",
	"responseHeader": {
	...
	...
	}
	}</pre>
		NIL	No

Response status	202	202	Yes

When we evaluate this closely below are points based on which we can decide the approach:

1. We have a different set of **Headers**, **API endpoints**, **Request Body**, and **Response body**. So, we cannot use the exact same method, but we can make some modifications to make it reusable.
2. **Request type** and **Response status** are the same.

How to Compare and Adapt

Below is a sample of the **Individual Disbursement** API call using Java programming language. The **createDisbursementTransaction** function is created from the API request. The structure of a typical SDK function created from a comparable API call is seen in the following code sample.

```
1  public AsyncResponse createDisbursementTransaction() throws MobileMoneyException {
2      this.clientCorrelationId = UUID.randomUUID().toString();
3
4      if (this.transaction == null) {
5          throw new MobileMoneyException(
6                  new HttpErrorResponse.HttpErrorResponseBuilder(Constants.VALIDATION_ERROR_CATEGORY,
7                          Constants.VALUE_NOT_SUPPLIED_ERROR_CODE)
8                                  .errorDescription(Constants.TRANSACTION_OBJECT_INIT_ERROR).build());
9      }
10
11     String resourcePath = API.TRANSACTION_TYPE.replace(Constants.TRANSACTION_TYPE, TransactionType.DISBURSEMENT);
12     MobileMoneyContext.getContext().getHTTPHeaders().put(Constants.CORRELATION_ID, this.clientCorrelationId);
13
14     return createRequest(HttpMethod.POST, resourcePath, this.transaction.toJSON(), notificationType, callBackURL,
15             AsyncResponse.class);
16 }
```

The values to be passed in as the **request body** are stored in the variable **transaction** in the snippet of code above (line 4) and is checked before the request is sent to make sure it is instantiated. The API endpoint is defined in a constant class and is obtained and added to the **resourcePath** variable (line 11). In the end, we call the **createRequest** method after all the necessary arguments have been verified and return it as a **AsyncResponse.class** type. With the code snippet above, let's see what we can modify to use it with the new API.

- **Function name**: The function name can be kept the same if needed or a new name can be given.
- **Headers:** Apart from **Authorization**, **X-Callback-Url**, **and Content-Type** all the other headers are different for the new API.
    - **Authorization** contains the **Bearer Token** required for authentication.
    - **X-Callback-Url** contains the **callback URL**, used for receiving the final response of the API call.
    - **Content-Type** is for specifying the type of data passed in.
    - **X-Reference-Id** is the same as the **clientCorrelationId**, a variable responsible for holding the **UUID** which is carried by the header **X-CorrelationID**. In this case, we can use the same for **X-Reference-Id** which is similar in usage in both scenarios. We only need to add the new **X-Reference-Id** header to the constant class and use it in the function.

    - **Ocp-Apim-Subscription-Key** is for providing access to this API. It is used in conjunction with the **Authorization** header. **Ocp-Apim-Subscription-Key** is different for each user and each use case.
    - **X-Target-Environment** specifies whether the Mobile Money API is a sandbox, production, staging or any other kind of environment.
- **Request Body:** The **transaction** object holds the request body of the API call. We can replace the **transaction** object with a new model class that contains our new request body and then make the necessary validations.
- **API URL:** The **resourcePath** variable will contain the API URL which is fetched from a predefined constant class. We can update this constant class with our new API URL.
- **Response Body:** In MMAPI, we get a detailed **response body** from where a **payload** is extracted and mapped to **AsyncResponse** class object as a response. In the comparing API, there is no response body. So we can give a standard response based on the response code received (**202**).

Once all the required parameters are validated, the **createRequest** method can be called. This method passes in all the parameters to the next phase where we execute the API call.

Use Case 2: Obtain a Disbursement Organisation Balance

In the **Disbursement Balance** scenario, a **GET** request is made to the API URL **/accounts/{identifierType}/{identifier}/balance** without any request body. The response will be an object of the model class **Balance**.

	MMAPI	Different API	Similar
API URL	/accounts/{identifierType}/{identifier}/balance	/disbursement/v1_0/account/balance	No
Headers	Authorization and X-API-Key	Authorization, Ocp-Apim-Subscription-Key and X-Target-Environment	No
Request type	GET	GET	Yes
Request body	NIL	NIL	

Response body	{
	"accountStatus": "available",
	"currentBalance": "1000000000.00",
	"availableBalance": "0.00",
	"reservedBalance": "0.00",
	"unclearedBalance": "0.00",
	"currency": "GBP"
	}
		{
		"availableBalance": "string",
		"currency": "string"
		}
			No

Response status	200	200	Yes

When we evaluate this closely below are points based on which we can decide the approach:

1. We have a different set of **Headers**, **API URL**, **Request Body,** and **Response body**. So, we cannot use the exact same method, but we can make some modifications to make it reusable.
2. **Request type** and **Response status** are the same.

How to Compare and Adapt

Following is the java code snippet of **Disbursement Balance** API call. The API call is converted into the method named **viewAccountBalance**. The following code snippet contains a typical structure of how a similar API call is converted into a method in SDK.

```
1  public Balance viewAccountBalance(Identifiers identifiers) throws MobileMoneyException {
2     if (identifiers == null) {
3        throw new MobileMoneyException(
4           new HttpErrorResponse.HttpErrorResponseBuilder(Constants.VALIDATION_ERROR_CATEGORY,
5                                                          Constants.VALUE_NOT_SUPPLIED_ERROR_CODE)
6                                 .errorDescription(Constants.IDENTIFIER_OBJECT_INIT_ERROR)
7                                 .build());
8     }
9
10    return createRequest(HttpMethod.GET, getResourcePath(API.ACCOUNT_BALANCE_REQUEST, identifiers), Balance.class);
11 }
```

In the above code snippet, variable **identifiers** which are passed in as function parameters (line 1) act as the **path parameter** of the **API URL**. The **identifiers** parameter is validated before executing the request (line 2). The **getResourcePath** function (line 10) concatenates the existing **API URL** with the passed in identifiers parameter to generate the correct URL. Once all the required parameters are validated, we call the **createRequest** method (line 10). This **createRequest** method passes in all the parameters to the next phase where we execute the API call. The last parameter, **Balance.class** (line 10) is the response body. It acts as the return type of the **createRequest** method.

With the code snippet above, let's see what we can modify to use it with the new API.

- **Function name**: The function name can be kept the same if needed or a new name can be given.
- **Headers:** Apart from **Authorization** all the other headers are different for the new API.
  - **Authorization** contains the Bearer Token required for authentication.
  - **Ocp-Apim-Subscription-Key** is for providing access to this API. It is used in conjunction with the **Authorization** header. **Ocp-Apim-Subscription-Key** is different for each user and each use case.
  - **X-Target-Environment** is based on the environment. It specifies whether it is a sandbox or any of the production environments.
- **Request Body:** No request body is passed in for both API calls.
- **API URL:** Since there are no parameters passed in for the new API, we don't need to generate the URL with any parameters. We can directly update the constant class that contains all the API URLs and use the new API URL here.

- **Response Body:** In MMAPI and the new API, we are getting different response bodies. We can create a new model class based on the response of the new API and replace the **Balance.class** with our new model class.

Once all the required parameters are validated, we can call the **createRequest** method. This method passes in all the parameters to the next phase where we execute the API call.

4. Adapting GSMA Mobile Money API SDKs

4.1. Code and Folder Structure

The structure presented below is a general description of how the MMAPI SDKs source codes are organised in the GitHub repositories.

4.1.1. General structure of the repository

The root filesystem organisation (files and folders) of an MMAPI SDK repository is presented below. It illustrates the type of files and folders that are mandatory and present in an SDK repository and what that file or folder holds in terms of code or documentation. Bear in mind that names can be different depending on the programming language you are using.

```
Root (/)
├── README.md
├── code-snippets
├── docs
├── samples
├── src
└── tests
```

- **`README.md`**: detailed description of your repository, with instructions on how to build and run the SDK, links to documentation, samples and/or code snippets, instructions on how to test the SDK and any other information that may be used by the developers to have an initial understanding on how to use the SDK.
- **`code-snippets`**: stores code snippets of the SDK demonstrating how to use it. These code snippets can be added to the developer's portal of the SDKs.
- **`docs`**: detailed documentation of the SDK, including all scenarios, authentication, error handling, and any other information needed by the developers.
- **`samples`**: fully functional example where the developer can run and see the SDK working. It is divided by use cases and scenarios.
- **`src`**: source code of the SDK, organised by one folder for each use case (accountlinking, billpayment, disbursement, internationaltransfer, merchantpayment, p2ptransfer, recurringpayment).

- **`tests`**: set of unit and integration tests for the SDK, with full coverage.

4.1.2. SDK source code organisation and description

Each module inside the source code folder contains the respective request class (**`accountlinking`** package contains **`AccountLinkingRequest`**, **`agentservices`** package **`AgentServiceRequest`**, etc.) which holds all the API calls. You can also find the **`Model`** classes (**`Account`**, **`Commission`**, **`Identity`**, etc.) that are required for that API request in the **`Model`** package.

```
accountlinking
├── model
│   └── Link.java
└── request
    └── AccountLinkingRequest.java
agentservices
├── model
│   ├── Account.java
│   ├── Commission.java
│   └── Identity.java
└── request
    └── AgentServiceRequest.java
```

If some API requests are common among all the modules, then they are added to the **`common`** package, which contains all the common API requests, models, and constants.

```
common
├── constants
│   ├── Environment.java
│   └── NotificationType.java
├── model
│   ├── AccountHolderName.java
│   ├── AccountIdentifier.java
│   ├── AsyncResponse.java
│   ├── AuthorisationCode.java
│   ├── Balance.java
│   ├── CustomData.java
│   ├── Fees.java
│   ├── Filter.java
│   ├── Identifiers.java
│   ├── MetaData.java
│   ├── Name.java
│   ├── PatchData.java
...
│   ├── Reversal.java
│   ├── ServiceAvailability.java
│   ├── Transaction.java
```

```
│   ├── TransactionFilter.java
│   └── Transactions.java
└── request
    ├── AuthorizationCodeRequest.java
    ├── CommonRequest.java
    ├── CreateTransactionRequest.java
    ├── TransferRequest.java
    └── ViewTransactionRequest.java
```

The **base** folder contains the core structure of the SDK, handling initialization, default values, HTTP connection handlers, task execution and miscellaneous activities related to the instantiation of the SDK.

```
base
├── APIManager.java
├── ConfigManager.java
├── ConnectionManager.java
├── DefaultHttpConnection.java
├── ExecuteTask.java
├── HttpConfiguration.java
├── HttpConnection.java
├── HttpResponse.java
├── SDKUtil.java
├── SSLUtil.java
├── constants
├── context
├── exception
├── model
└── util
```

Inside this **base** package, the **context** is the main package where we can find the **MobileMoneyAuthentication**, **MobileMoneyContext** and the **MMClient** classes which handle all the API requests and manage the singleton instance.

```
base/context
├── MMClient.java
├── MobileMoneyAuthentication.java
└── MobileMoneyContext.java
```

Furthermore, the **exception** package contains all the custom exceptions written to handle any incoming exceptions from MMAPI in case of any error or any other anticipated exceptions that can occur during the working of the SDK.

```
base/exception
├── BaseException.java
├── MobileMoneyException.java
├── SSLConfigurationException.java
└── UnauthorizedException.java
```

The **`model`** package contains all the models for the exception classes.

```
base/model
├── HttpErrorMetaData.java
├── HttpErrorResponse.java
└── SDKResponse.java
```

The **`util`** package contains all the utility classes that help reduce redundancy. For example, converting JSON strings to Objects or null value checking are defined here.

```
base/util
├── EnumUtils.java
├── JSONFormatter.java
├── ResourceLoader.java
├── ResourceUtils.java
└── StringUtils.java
```

4.1.3. SDK samples organisation

The **`samples`** folder has sample codes of various use cases and scenarios of the MMAPI. It is provided to demonstrate how to use the SDK in an application. It is organised the same way as the SDK source code: each scenario has a folder where it is stored the source code of the sample codes. Below is the organisation of the folders and files for the SDK samples.

```
samples
└── src
    ├── accountlinking
    │   ├── CreateAccountLink.java
    │   ├── CreateReversal.java
    │   ├── CreateTransferTransaction.java
    │   ├── ViewAccountBalance.java
    │   ├── ViewAccountLink.java
    │   ├── ViewAccountTransactions.java
    │   ├── ViewRequestState.java
    │   ├── ViewResponse.java
    │   ├── ViewServiceAvailability.java
    │   └── ViewTransaction.java
    ├── base
    │   └── SDKClient.java
    ├── agentservice
    ├── billpayment
    ├── disbursement
    ├── internationaltransfer
    ├── merchantpayment
    ├── p2ptransfer
    └── recurringpayment
```

For each use case folder, you have the source code files containing a fully functional application that represents a specific scenario inside the use case. Each scenario is implemented using the SDK guidelines and code for the purpose of demonstrate how to use the SDK in an application. For example, in **`accountlinking`** folder, the file **`CreateAccountLink.java`** represents the sample code that uses the SDK to create a account link, according to the MMAPI Specification.

4.1.4. SDK documentation

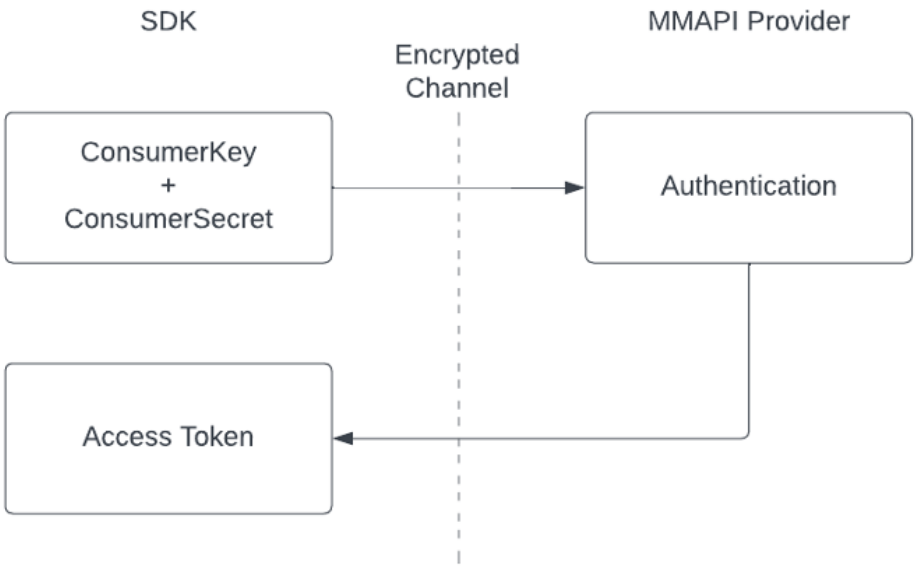
The **`docs`** folder contains the documentation on how to use the SDK. It is written based on the Markdown Language (e.g. .md extension) to be processed and formatted by GitHub and available for read directly when browsing the repository. The documentation is based on demonstrating how the SDK should be instantiated and integrated in applications using code snippets and samples as examples. The `docs` folder contains any documentation you may have for the SDK you are developing and can contain other types of documents, like design strategies, flow diagrams, security patterns, to name a few, and in any format like PDF, word documents, photos, pictures, etc. Below you can see the file organisation of the documentation in Markdown Language, organised as one Markdown file per scenario inside a use case.

```
docs
├── accountLinking
│   ├── createAccountLink.Readme.md
│   ├── createReversal.Readme.md
│   ├── createTransferTransaction.Readme.md
│   ├── viewAccountBalance.Readme.md
│   ├── viewAccountLink.Readme.md
│   ├── viewAccountTransactions.Readme.md
│   ├── viewRequestState.Readme.md
│   ├── viewResponse.Readme.md
│   ├── viewServiceAvailability.Readme.md
│   └── viewTransaction.Readme.md
├── agentService
├── billPayment
├── disbursement
├── internationalTransfer
├── merchantPayment
├── p2pTransfer
└── recurringPayment
```

4.2. API authentication SDK

Authentication is an essential step on the initialisation and usage of the SDK. In most Mobile Money environments, customers are required to register and obtain a set of keys, often called as

`ConsumerKey`, **`SubscriptionKey`**, **`CustomerKey`**, etc. In some instances, additional keys or parameters such as a **`ReferenceID`** or **`API Key`** may also be necessary.
The SDKs developed for MMAPI uses the **`ConsumerSecret`** and **`ConsumerKey`** values as the credentials of the customer to authenticate via **`OAuth2`** and receive an **`Access Token`**. Additionally, a third value called **`API Key`** is also needed for every request made to the MMAPI using the **`Access Token`** generated after authentication. Below is depicted the authentication flow used by the SDK.



For the SDK be able to make request in MMAPI, an **Access Token** is required. It is created through an object of the **MMClient** class using the **MobileMoneyAuthentication** clas, following the authentication flow of the MMAPI. The **MMClient** class in turn connects with the **MobileMoneyContext** class, which is a singleton class that handles all the requests in the background and stores the **Access Token** for future usages. If the **Access Token** expires, then the **MobileMoneyContext** class communicates with **MobileMoneyAuthentication** class for creating a new **Access Token** as a background operation without interrupting the flow of the application.

Let's consider the sample **CreateReversal.java** under **accountlinking** package from **mmapi-java-sdk-samples** project to understand how authentication is done in MMAPI.

```
1  public class CreateReversal extends SDKClient {
2      public static void main(String... args) {
3          try {
4              MMClient mmClient = new MMClient(get("CONSUMER_KEY"), get("CONSUMER_SECRET"), get("API_KEY"));
5              AccountLinkingRequest accountLinkingRequest = new AccountLinkingRequest();
6
7              Reversal reversal = new Reversal();
8              reversal.setType("reversal");
9              accountLinkingRequest.setReversal(reversal);
10
```

```
11          String transactionReference = "REF-1635251574104";
12
13          System.out.println("Please wait...");
14          AsyncResponse sdkResponse = mmClient.addRequest(accountLinkingRequest).addCallBack(get("CALLBACK_URL"))
15                  .createReversal(transactionReference);
16
17          System.out.println(String.format("Transaction Reversal Status: %s", sdkResponse.getStatus()));
18      } catch (MobileMoneyException ex) {
19          System.out.println(String.format("Mobile Money Exception: %s", ex.getError().getErrorDescription()));
20      }
21  }
22 }
```

The **MMClient** object is initialized with a set of predefined values required for authentication (line 4). But this only creates an object of **MMClient** and doesn't make any API calls to initiate authentication.

Later the **addRequest** method on the **MMClient** object (line 14) is called and then the authentication flow happens, resulting in the generation of the Access Token.

Inside the **addRequest** method of **MMClient** class, the method **createContext** checks if the variable **instance** is already instantiated (line 10 and line 12). If not, then a new **instance** object is created (line 13). In this case, the **instance** variable is an object of **MobileMoneyContext** and acts as a singleton design pattern. Below is the **createContext** method definition.

```
1  public static void createContext(
2      String consumerKey,
3      String consumerSecret,
4      String apiKey,
5      Environment mode,
6      String callBackUrl,
7      SecurityLevel securityLevel,
8      Map<String, String> configurations
9  ) {
10     if (instance == null) {
11         synchronized (MobileMoneyContext.class) {
12             if (instance == null) {
13                 instance = new MobileMoneyContext(
14                     consumerKey,
15                     consumerSecret,
16                     apiKey,
17                     mode,
18                     callBackUrl,
19                     securityLevel,
20                     configurations
21                 );
22             }
23         }
24     }
25     instance.callBackUrl = callBackUrl;
26 }
```

Ultimately, the **createContext** method relies on the constructor of the class **MobileMoneyContext** to handle the authentication. The **Access Token** is generated there, by calling the method **getAccessToken** (line 18) of the **MobileMoneyAuthentication** class, passing in the required parameters, in this case, the **Consumer Key**, **Consumer Secret** and **API Key** (line 10). Below is the code for the constructor **MobileMoneyContext**.

```
1  private MobileMoneyContext(
2      String consumerKey,
3      String consumerSecret,
4      String apiKey,
5      Environment mode,
6      String callBackUrl,
7      SecurityLevel securityLevel,
8      Map<String, String> configurations
9  ) {
10     this.credential = new MobileMoneyAuthentication(consumerKey, consumerSecret, apiKey);
11     if (configurations != null && configurations.size() > 0) {
12         this.credential.addConfigurations(configurations);
13     }
14     this.setMode(mode);
15     this.setSecurityLevel(securityLevel);
16     this.callBackUrl = callBackUrl;
17     try {
18         this.credential.getAccessToken();
19         instance = this;
20     } catch (Exception e) {
21     }
22 }
```

The most important part of the code is the object **credential**. As the authentication is handled as a singleton design pattern, after all the steps taken to authenticate and generate the **Access Token**, whenever we make a call using the method **mmClient.addRequest(…)**, we'll be getting the same **instance** of the object **credential**. This means that no new authentication flow is triggered inside by the SDK and the credential object returned will always contain the most up-to-date and valid **Access Token**.

If the **Access Token** expires, the **credential** instance will not hold a valid **Access Token** anymore. In this case, the **getRefreshToken** method in **MobileMoneyContext** class starts the process of get an updated **Access Token** value by calling **getRefreshToken** method of **MobileMoneyAuthentication** class.

4.3. Adapting an API Scenario into an API SDK

4.2.1. Explanation of the method/function structure

For converting an API request into a method, we must understand the nature/requirements of that API call, i.e., Headers, HTTP Method, Request Body, Response Body. MMAPI has different modules such as **accountlinking**, **billpayment**, **disbursement**, **internationaltransfer**, **merchantpayment**, **p2ptransfer** and **recurringpayment** and each module contains the methods for their respective API calls. Common API calls are written as methods in **common** classes that are shared among all the modules.

A typical MMAPI call when converted into an SDK function will have the following structure:

- **UUID generator:** If the request contains **X-CorrelationID** as a header, then we will generate a UUID inside that function and pass it as the **X-CorrelationID** header for that API call.

- **Path parameters:** If the request contains path parameters in the API URL, then we structure the function, so that these path parameters can be passed into the function as the function parameters. The parameters need to be validated inside the function before making the API call.
- **Request Body:** If the request contains a request body, then we create a model class with all the fields of the request body. The model class needs to be validated inside the function before making the API call.
- **API URL:** The API URL of the API will be fetched from a constant class API.class. This class should contain all the API URLs. If there are path parameters in the API URL, then we will have to generate the URL with the parameters passed in, e.g. the **getResourcePath** function found in **ResourceUtils** class generates the URL with the help of parameter.
- **Return type:** The return type of the function will be decided based on the response body of the API call. A model class need to be created based on the response body fields.
- **Forwarding the request:** At the end the request will be made by calling the **createRequest** function by passing in all the required parameters. The authentication processes will be handled by the **MobileMoneyAuthentication** class.

4.2.2. How you could adapt an SDK

Let's start by explaining how an SDK function would be triggered by the end user and how this function has been structured. Following is an example on how the **createAccountLink** method of **AccountLinkingRequest** class can be called in an application.

In the example below, at line number 2, we are initializing an **MMClient** object, which will be used for making API calls. Starting at line 3, we created an **AccountLinkingRequest** object to make the **createAccountLink** method call. Then from lines 5 to 28, we are initializing and feeding all the necessary values required to make the API call.

In this specific example of **createAccountLink** method, two separate Model classes are passed in. They are **Link** and **Identifiers**. Here **Link** is the request body, which is set using the **setLink** method, while **Identifiers** Model is a list of **AccountIdentifier** classes fed in as **List<AccountIdentifier>**. This **Identifiers** class is the path parameter for the **createAccountLink** API URL, and is passed in as the method parameters.

```
1  try {
2     MMClient mmClient = new MMClient(get("CONSUMER_KEY"), get("CONSUMER_SECRET"), get("API_KEY"));
3     AccountLinkingRequest accountLinkingRequest = new AccountLinkingRequest();
4
5     List<AccountIdentifier> sourceAccountIdentifiers = new ArrayList<>();
6     RequestingOrganisation requestingOrganisation = new RequestingOrganisation();
7     List<CustomData> customDataList = new ArrayList<>();
8
9     sourceAccountIdentifiers.add(new AccountIdentifier("accountid", "2999"));
10    customDataList.add(new CustomData("keytest", "keyvalue"));
11    requestingOrganisation.setRequestingOrganisationIdentifierType("organisationid");
12    requestingOrganisation.setRequestingOrganisationIdentifier("testorganisation");
13
14    Link link = new Link();
15    link.setSourceAccountIdentifiers(sourceAccountIdentifiers);
```

```
16    link.setMode(Mode.BOTH.getMode());
17    link.setStatus(Status.ACTIVE.getStatus());
18    link.setRequestingOrganisation(requestingOrganisation);
19    link.setRequestDate("2018-07-03T11:43:27.405Z");
20    link.setCustomData(customDataList);
21
22    accountLinkingRequest.setLink(link);
23
24    List<AccountIdentifier> identifierList = new ArrayList<>();
25    identifierList.add(new AccountIdentifier("accountid", "15523"));
26
27    AsyncResponse sdkResponse = mmClient.addRequest(accountLinkingRequest)
28          .createAccountLink(new Identifiers(identifierList));
29
30    System.out.println(String.format("Account Link Creation Status: %s", sdkResponse.getStatus()));
31  } catch (MobileMoneyException ex) {
32    System.out.println(String.format("Mobile Money Exception: %s", ex.getError().getErrorDescription()));
33  }
```

The API request is made at line number 27. The first method call of the API request – **mmClient.addRequest(accountLinkingRequest)** – creates an access token if it is not yet generated. After that, the second method call – **createAccountLink(new Identifiers(identifierList))** – is made on the **accountLinkingRequest** object. This **accountLinkingRequest** object can further be used to make any requests that belong to the **accountLinking** module, for example, **viewAccountLink**, **createTransferTransaction**, etc.

One important point to note is that **AccountLinkingRequest** extends the **ViewTransactionRequest** class. This **ViewTransactionRequest** class holds several common **Transaction type requests** methods such as **viewAccountTransactions, viewTransaction, createReversal,** etc. These request methods are available to **AccountLinkingRequest** or any other **Request** classes like **DisbursementRequest, RecurringPaymentRequest,** etc., since these classes inherit the **ViewTransactionRequest**.

Following is the code for the API call **createAccountLink** method. From here onwards, the SDK handles everything.

```
1  public AsyncResponse createAccountLink(Identifiers identifiers) throws MobileMoneyException {
2     this.clientCorrelationId = UUID.randomUUID().toString();
3     if (identifiers == null) {
4        throw new MobileMoneyException(
5              new HttpErrorResponse.HttpErrorResponseBuilder(Constants.VALIDATION_ERROR_CATEGORY,
6                    Constants.VALUE_NOT_SUPPLIED_ERROR_CODE)
7                          .errorDescription(Constants.IDENTIFIER_OBJECT_INIT_ERROR).build());
8     }
9     if (link == null) {
10       throw new MobileMoneyException(
11             new HttpErrorResponse.HttpErrorResponseBuilder(Constants.VALIDATION_ERROR_CATEGORY,
12                   Constants.VALUE_NOT_SUPPLIED_ERROR_CODE)
13                         .errorDescription(Constants.ACCOUNT_LINK_OBJECT_INIT_ERROR).build());
14    }
15    String resourcePath = getResourcePath(API.CREATE_ACCOUNT_LINKS, identifiers);
16    MobileMoneyContext.getContext().getHTTPHeaders().put(Constants.CORRELATION_ID, this.clientCorrelationId);
17    return createRequest(HttpMethod.POST, resourcePath, link.toJSON(), notificationType, callBackURL,
18          AsyncResponse.class);
19  }
```

At line 2, we have created a new **UUID** which will be passed in as the **clientCorrelationId**. Then a series of validation checks are made at lines 3 through 14 on the objects required by the Model – **identifiers** and **link**. An **MobileMoneyException** throws the corresponding exception to the user if any validation of these objects fails.

At line 15, we are fetching the **API URL** from the list of constants (**API.CREATE_ACCOUNT_LINKS**) and formatting it based on the **identifierList** to generate the final URL for the API Request, then we store it to the variable **resourcePath**. On the line 16, we are updating the headers in the **MobileMoneyContext** by adding **clientCorrelationId** to the headers.

And finally, we are making a call to **createRequest** method by passing in all the necessary parameters:

- **HttpMethod.POST** – define what is the HTTP Method of the request
- **resourcePath** – final API URL for the request
- **link.toJSON()** – **link** object is being converted to JSON for passing in as the request body
- **notificationType** – **notificationType** decides if the request is of type **POLLING** or **CALLBACK**
- **callBackURL** – **callBackURL** is where we receive the response if the **notificationType** is set as **CALLBACK**
- **AsyncResponse.class** – **AsyncResponse** is the return type of this particular API call

API requests are made by calling **createRequest** method of the **ResourceUtils** class. Since it is an important method, below we are showing its source code and explaining it.

```
1  protected <T> T createRequest(HttpMethod httpMethod, String resourcePath, String payLoad,
2        NotificationType notificationType, String callBackURL, Class<T> responseObject)
3        throws MobileMoneyException {
4     T sdkResponse = null;
5
6     HttpResponse requestResponse = requestExecute(httpMethod, resourcePath, payLoad, notificationType, callBackURL);
7     if (requestResponse.getPayLoad() instanceof String) {
8        sdkResponse = JSONFormatter.fromJSON((String) requestResponse.getPayLoad(), responseObject);
9     }
10
11     return sdkResponse;
12  }
```

At line 6, the **requestExecute** method is the one who executes a HTTP Request with the parameters and returns a response to the **HttpResponse** object. This **HttpResponse** will contain a payload that is returned (line 7) and formatted as a JSON (line 8).
The **requestExecute** method contains many lines of code to manage the incoming API request, so we are showing only a small part of the method source code that is necessary to explain its logic.

```
1  protected static HttpResponse requestExecute(HttpMethod httpMethod, String resourcePath, String payLoad,
2        NotificationType notificationType, String callBackURL) throws MobileMoneyException {
3     HttpResponse responseData = null;
4     MobileMoneyContext apiContext = MobileMoneyContext.getContext();
```

```
5   Map<String, String> cMap;
6   Map<String, String> headersMap;
7
8   if (apiContext != null) {
9       String accessToken = apiContext.fetchAccessToken();
10      if (accessToken == null) {
11          throw new IllegalArgumentException(Constants.EMPTY_ACCESS_TOKEN_MESSAGE);
12      }
13
14      if (apiContext.getHTTPHeader(Constants.HTTP_CONTENT_TYPE_HEADER) == null) {
15          apiContext.addHTTPHeader(Constants.HTTP_CONTENT_TYPE_HEADER, Constants.HTTP_CONTENT_TYPE_JSON);
16      }
17
18      ...
19
20      headersMap = apiContext.getHTTPHeaders();
21      headersMap.put(Constants.AUTHORIZATION_HEADER, Constants.BEARER + accessToken);
22      APIManager apiManager = new APIManager(cMap, headersMap);
23      apiManager.setResourcePoint(resourcePath);
24      HttpConfiguration httpConfiguration = getHttpConfiguration(httpMethod, apiManager);
25      responseData = executeWithRetries(apiContext, () -> execute(apiManager, httpConfiguration));
26
27      if (responseData == null || responseData.getPayLoad() == null) {
28          throw new MobileMoneyException(
29                  new HttpErrorResponse.HttpErrorResponseBuilder(Constants.INTERNAL_ERROR_CATEGORY,
30                          Constants.GENERIC_ERROR_CODE).errorDescription(Constants.GENRAL_ERROR).build());
31      } else if (!responseData.isSuccess() && responseData.getPayLoad() instanceof String) {
32          HttpErrorResponse errorResponse = JSONFormatter.fromJSON((String) responseData.getPayLoad(),
33                  HttpErrorResponse.class);
34          throw new MobileMoneyException(errorResponse);
35      }
36  }
37
38  return responseData;
39 }
```

In the **requestExecute** method, we first check if the **apiContext** is null (line 8). If the context is valid, we check if the **access token** and **headers** are valid (lines 10 through 16). Once everything is verified, we set the **AUTHORIZATION_HEADER** and **resourcePath** (lines 20 through 24).

At line number 25, we execute the API request through the **executeWithRetries** method and store the result in the variable **responseData**, which is checked for errors of failures (lines 27 through 34), raising the appropriated **MobileMoneyException** to the user if any. In the end, the actual **responseData** variable is returned as the result of the method **requestExecute**.

It is possible to convert other API requests to methods following the above guidelines and examples.

5. Testing your SDK

5.1. Unit Testing

The purpose of SDK unit tests is to isolate each SDK scenario and perform tests on it, ensuring that the functionality implemented by that scenario is working as expected. Let's consider an example where we are writing a unit test for **createAccountLink** scenario. Following is a sample unit test written in Java with Junit.

```
1  @Test
2  @DisplayName("Create Account Link Success")
3  void createAccountLinkTestSuccess() throws MobileMoneyException {
4      AsyncResponse expectedSdkResponse = getAsyncResponse();
5
6      AccountLinkingRequest accountLinkingRequest = new AccountLinkingRequest();
7
8      List<AccountIdentifier> identifierList = new ArrayList<>();
9      identifierList.add(new AccountIdentifier("accountid", "15523"));
10     Identifiers identifiers = new Identifiers(identifierList);
11
12     AccountLinkingRequest accountLinkingRequestSpy = Mockito.spy(accountLinkingRequest);
13
14     Mockito.doReturn(expectedSdkResponse).when(accountLinkingRequestSpy).createAccountLink(identifiers);
15
16     AsyncResponse actualSdkResponse = accountLinkingRequestSpy.createAccountLink(identifiers);
17
18     assertNotNull(expectedSdkResponse);
19     assertNotNull(actualSdkResponse);
20     assertEquals(expectedSdkResponse.getServerCorrelationId(), actualSdkResponse.getServerCorrelationId());
21     assertEquals(expectedSdkResponse.getStatus(), actualSdkResponse.getStatus());
22 }
```

The idea of a unit test is comparing the SDK response with an expected response. This is done in the code above by creating the **expectedSdkResponse** object (line 4) and feed it with specific known data that we will compare with the actual data stored by the **actualSdkResponse** object (line 16). Later, we will test if the **expectedSdkResponse** and **actualSdkResponse** data match (lines 18-21). The values that we are using in the unit test are defined on lines 8-10.

We used the **Mockito framework** to implement the unit tests. We create a **mock call** of **accountLinkingRequest** to the **createAccountLink** method using the **spy** concept of the **Mockito framework** (lines 12-14). We also create a request to get the actual values returned by the method in test on line 16. After receiving the **actualSdkResponse** value, we use different types of **assert** methods to check if the response is valid (lines 18-21).

Similarly, it is possible to create unit tests for all the use cases and scenarios implemented to reach a full coverage of the SDK.

5.2. Integration Testing

The purpose of SDK integration tests is to validate the SDK by performing tests on a scenario where multiple different methods and components are used, ensuring that the workflow of that scenario is working as expected. Integration tests often uses external system, as this is the case for this **createAccountLink** integration test, which uses the MMAPI simulator as a server to validate the API calls, scenarios and uses cases.

Let's consider an example where we are writing an integration test for **createAccountLink** scenario. Following is a sample integration test written in Java with Junit.

```
1  @Test
2  @DisplayName("Create Account Link Success")
3  void createAccountLinkTestSuccess() throws MobileMoneyException {
4      MMClient mmClient = new MMClient(loader.get("CONSUMER_KEY"),
5                              loader.get("CONSUMER_SECRET"),
6                              loader.get("API_KEY"))
7              .addCallBackUrl(loader.get("CALLBACK_URL"));
8
9      AccountLinkingRequest accountLinkingRequest = new AccountLinkingRequest();
10     accountLinkingRequest.setLink(getLinkSuccessObject());
11     List<AccountIdentifier> identifierList = new ArrayList<>();
12     identifierList.add(new AccountIdentifier("accountid", "15523"));
13     AsyncResponse sdkResponse = mmClient.addRequest(accountLinkingRequest)
14             .createAccountLink(new Identifiers(identifierList));
15
16     assertNotNull(sdkResponse);
17     assertNotNull(sdkResponse.getServerCorrelationId());
18     assertTrue(Arrays.asList("pending", "completed", "failed").contains(sdkResponse.getStatus()));
19     assertEquals(sdkResponse.getNotificationMethod(), "callback");
20
21 }
```

The integration test for **createAccountLink** is very similar to the unit test for the same scenario, but the main difference is that it uses an external MMAPI server to validate the requests. This is done when the integration test creates a **mmClient** object, which handles the authentication to a MMAPI server using valid and real data (lines 4-7). Note that those real values for authentication, like **ConsumerKey**, **ConsumerSecret** and **API Key**, and parameters to handle responses, like **Callback URL** are passed to the initialization of the **mmClient** object.

You can notice also that the **Mockito framework** used in the unit tests to mock requests and responses are not used in the integration tests, because the integration test code needs to be fully functional as if it was in a real application, thus using real data. The code representing the full scenario for the **createAccountLink** is on lines 9-14.

It can be difficult to know beforehand the values returned by a real MMAPI server, so the assertions focus on check if a pattern or known values of the MMAPI specification are present in the response. For example, the MMAPI specification has the following status response values – pending, completed, and failed – so in this case the integration test can assert for those values. Other types of assertions are just to check if everything is present in the request without checking for specific values, as this is the case for check if the **ServerCorrelationId**. The assertions for the **createAccountLink** integration test can be found on lines 16-19.

6. Documentation and Distribution

When creating documentation for the SDK, it is essential for providing the developers with a clear understanding of the SDK's purpose, features, and usage. Refer the documentation of the MMAPI sdk for more details https://github.com/gsmainclusivetechlab/mmapi-java-sdk.

Here are some best practices to consider when writing a documentation for an SDK:

- Table of Contents: Include a table of contents at the beginning of the documentation to help users quickly navigate through the document and find the information they need.
- Requirements: An SDK may have dependencies during its implementations. Point out each of the dependencies for the usage of the SDK. This helps developers from spending time on unnecessary debugging.
- Installation: Provide detailed instructions on how to install the SDK. Include information about any prerequisites, dependencies, or system requirements. Offer various installation methods, such as how to build the SDK, where to download it from, how to run it etc.
- Usage and Code Samples: Provide comprehensive documentation on how to use the SDK's main features and functionalities. Include code samples and examples for various API calls and use cases. Ensure that the code samples are clear, well-commented, and cover every scenario.
- Error Handling: Devote a section to explain how the SDK handles errors and how developers can handle errors effectively in their applications.
- Use cases: Consider providing links to each use case and user scenarios. This will allow developers to dive deeper into the API's details.
- Explain test cases: Explain how tests are written and how the developer can utilize the tests.
- Samples: Provide a sample project which includes each of the use cases and user scenarios.

7. Conclusion

In conclusion, creating a successful and effective SDK requires a careful balance of technical expertise, user-centric design, and meticulous attention to detail. As developers, you hold the power to shape the experiences of countless others who will use your SDK, integrating it into their projects and applications. By following the steps outlined in this guide, you're well-equipped to use the GSMA SDKs to support you to navigate the complex landscape of SDK development.

Studying this guide and examining the corresponding sample SDK code on GitHub, you will obtain a thorough understanding of the process involved in creating an SDK for any Mobile Money API. The documentation also provides guidance on evaluating your mobile money API and comparing it with MMAPI, allowing you to utilize the GSMA SDK repository and establish your own SDK.

Remember that a well-designed SDK not only provides a powerful toolset but also fosters a sense of trust and reliability with your users. Continuously seek feedback from developers who integrate your SDK, as this iterative process will lead to improvements and enhancements that can make

a substantial difference. Prioritize clear documentation, maintain a consistent update cycle, and consider the broader context in which your SDK will be used.

SDK development is not just about code; it's about building connections and enabling innovation. The mobile money team is committed to providing the support the mobile money ecosystem needs to ensure the quality of the assets developed. This enables not only a better innovation and development environment, but also better services for end users.

As you embark on this journey, keep in mind that your efforts contribute to a larger ecosystem, empowering fellow developers to create remarkable products and solutions. So, go forth with confidence, armed with the knowledge from this guide, and craft SDKs for your APIs that will inspire, simplify, and reshape the way technology is used across Mobile Money services. The creation of excellent tools will undoubtedly leave a lasting impact on the world of software development.