Viper integration:

For integrating viper with your application, Viper has several API for initiating, those are given below.

- Here userName is the name in the application
  ```
  ViperClient.on(context, userName)
  ```

- Here userImage is an index of the application profile
  ```
  ViperClient.on(context, userName, userImage)
  ```

- Here appToken is the token that will uniquely identify the application and currently Service using the application package name as appToken
  ```
  ViperClient.on(context, appToken, userName)
  ViperClient.on(context, appToken, userName, userImage)
  ```

You can choose one as per your parameter sequence

For stopping the mesh service or destroy the full mesh service then you can trigger the following API

```
viperClient.stopMesh();
viperClient.destroyMeshService()
```

And if you want to restart the mesh service or reset the viper SDK instance then call

```
viperClient.restartMesh();
viperClient.resetViperInstance()
```

Viper API's

Following API's you can access based on your features

When the profile information will be updated call the following API

```
viperClient.updateMyInfo(name)
viperClient.updateMyInfo(name, imageIndex)
```

When a user will update the user information and you receive the updated info then call the updateUserInfo API

```
viperClient.updateUserInfo(userId, name)
viperClient.updateUserInfo(userId, name, imageIndex)
```

For send a message to other nodes, then use sendMessage API with following parameters,

1. Sender id that is your Ethereum id,

2. Receiver Ethereum id [That you get during user discovery]
3. Message Id, that is a unique string, you can generate it using
   `UUID.randomUUID().toString()`
4. Data, that is raw byte array formate
5. Notification toggle, that indicates the receiver will through notification for this data or not.
   ```
   viperClient.sendMessage(senderId, receiverId, messageId, data,
   notificationToggle)
   ```

For send a file content, the following parameters needed and this sendFileMessage API return a content send id

1. Receiver Ethereum id
2. File directory path
3. Metadata is a byte array data that you can use for some information send during file send
   ```
   viperClient.sendFileMessage(receiverId, contentPath, metaData)
   ```

For content send if sender or receiver wants the resume, which content was failed then call the sendFileResumeRequest method

```
viperClient.sendFileResumeRequest(contentId, metaData)
```

During sending a file content TeleService maintains te content sends related particulars. After successfully send the file remove the information from the application,

```
viperClient.removeSendContent(contentId)
```

For ensuring a is connected or disconnected then call the API check connection status using user id and if the user is in offline then you informed by the user offline event

```
viperClient.checkConnectionStatus(friendsId)
```

For getting node connection type then you can call getLinkTypeById API, and it will return an integer value, that indicates the type. Describe bellow the value meaning

WIFI_ONLINE →	1
BLE_ONLINE →	2
WIFI_MESH_ONLINE →	3
BLE_MESH_ONLINE →	4
INTERNET_ONLINE →	5
HB_ONLINE →	6
HB_MESH_ONLINE →	7

```
viperClient.getLinkTypeById(nodeId)
```

For getting all sellers who have an active internet then you can use getInternetSellers API. It will provide a list of sellers Ethereum ids

```
        viperClient.getInternetSellers()
```

After initiating the Mesh Service, if the application has support to wallet creation using TeleService UI then call

```
        viperClient.openWalletCreationUI()
```

For accessing your wallet and using data buy and sell functionalities you can enter the service sided UI calling the following API

```
        viperClient.openDataPlanActivity()
        viperClient.openWalletActivity(avaterData)
```

N.B.- Here avatar data is your profile avatar image data, that is needed for populating the avatar on the wallet page.

If any interruption occurred for disabling permission from setting page then application informed by an event. For performing the mesh service need to trigger the allow permission API with missing permissions list and allow the permission from system popup

```
        viperClient.allowMissingPermission(permissions)
```

Viper Receiver event:

There are several events that viper received from service and provides the event's data to the client app. Now we are here discuss these types of events which are described below:

AppDataObserver: AppDataObserver is a class that receives and supply all data event in **viper** SDK. For receiving any event we have to use this class and call the below method:

```
        AppDataObserver.on().startObserver(EVENT_NAME, CONSUMER)
```

Here **EVENT_NAME** indicates the data types of events. For example, if a user discovers via SDK so in viper, we can get the information via **ApiEvent.PEER_ADD. ApiEvent** is a class that holds all event's names. **CONSUMER** is the actual listener of each event's data. If any data received in viper SDK then **CONSUMER** will be notified of its perspective event. We will discuss all event's names and all events data.

Events Name and Data:

ApiEvent.TRANSPORT_INIT: This is the mesh service initialization listener. When mesh service is initialized successfully or not then all information can be found from this event. The event data class is **TransportInit**.

Example:

```
        AppDataObserver.on().startObserver(ApiEvent.TRANSPORT_INIT, event ->
{
        TransportInit transportInit = (TransportInit) event;   });
```

The **TransportInit** has four properties.

1. *nodeId (String)*: This is user NodeId.
2. *publicKey (String):* The ethereum public key. That is also used for encryption and decryption.
3. *success (boolean):* The boolean value for success or error flag
4. *msg (String):* Information about success or error.

ApiEvent.PEER_ADD: If any user node is discovered via our **MeshService** then we can use the user node id from this event. And we can get the data from **PeerAdd** class.

Example:

```
        AppDataObserver.on().startObserver(ApiEvent.PEER_ADD, event -> {
                   PeerAdd peerAdd = (PeerAdd) event;    });
```

From the **PeerAdd** class, we can get discovered user nodeId.

ApiEvent.PEER_REMOVED: If any node disconnected from MeshService or removed from MeshService then viper SDK will notify by this event. And we can get the data from **PeerRemoved** class.

Example:

```
        AppDataObserver.on().startObserver(ApiEvent.PEER_REMOVED, event -> {
                 PeerRemoved peerRemoved = (PeerRemoved) event; });
```

From the **PeerRemoved** class, we can get the disconnected user nodeId.

ApiEvent.USER_INFO: This event is responsible for user data. We can say that user profile info. If the client app wants to show discovered user info in the view like name, then this kind of information can be found from this event. For sending USER_INFO during discovery, the information can be found from the upper description. And we can get data from **UserInfoEvent** class. As an example we can see:

Example:

```
        AppDataObserver.on().startObserver(ApiEvent.USER_INFO, event -> {
                UserInfoEvent userInfoEvent = (UserInfoEvent) event; });
```

The **UserInfoEvent** contains some information includes *address* (User nodeId), *avatar* (user avatar id that use in **Telemesh**), *username* (User display name), *regTime* (Registration time) etc. After receiving this event we can convert our own model from this **UserInfoEvent.**

ApiEvent.DATA: This is the most important and customizable event in viper SDK. For receiving any kind of data like Message or other information this event is used. For sending message data the brief description can be found from the **viper sender event section.** From this event, we can get data from **DataEvent** class. As an example we can see:

Example:

```
              AppDataObserver.on().startObserver(ApiEvent.DATA, event -> {
                     DataEvent dataEvent = (DataEvent) event;  });
```

The **DataEvent** class has three properties:

1. *peerId (String) :* It is the id of the sender node.
2. *data (byte[]):* This byte array contains the client apps data. It can be JSON or raw string. So client app can be able to convert the byte array data to its own model. For more information can be found from the **viper sender event** section.
3. *dataType (byte):* This is **1-byte** data that contains the type of data. It is part of the client app. During sending data client app can able to add **1-byte** information about its data so that the client app can understand which data is received. For example, we can say if **dataType** is 1 then the data will be TextMessage if the **dataType** is 2 the data will be broadcast message, etc. There are more pieces of information on how to send **1-byte** information can be found in **viper sender event** section

ApiEvent.DATA_ACKNOWLEDGEMENT: This event is used for getting acknowledgment events of sent data like data received or delivered etc. This data can be found in **DataAckEvent** class
For example, we can see:

Example:

```
        AppDataObserver.on().startObserver(ApiEvent.DATA_ACKNOWLEDGEMENT,
event -> {
                DataAckEvent dataAckEvent = (DataAckEvent) event;  });
```

The DataAckEvent has two properties:

1. *dataId(String)*: The **dataId** which is used to send Data.
2. *status (Int):* The status indicates Data is received or delivered

The status type can be found in **Constant.MessageStatus** class. This status are:

```
int SENDING = 1
int SEND = 2
int DELIVERED = 3
int RECEIVED = 4
int FAILED = 5
```

ApiEvent.WALLET_CREATION_EVENT: This event is used for observing wallet create or load status. When a client application runs then a new wallet created or an existing wallet loaded. So wallet load or create listener data can be found in this event and **WalletCreationEvent** class is the data container of this event.

Example:

```
        AppDataObserver.on().startObserver(ApiEvent.WALLET_CREATION_EVENT,
event -> {
           WalletCreationEvent walletCreationEvent = (WalletCreationEvent)
event;    });
```

WalletCreationEvent contains three properties

1. *successStatus (Boolean)*: Wallet loaded or not.
2. *nodeId (String):* Wallet ETH Id.
3. *statusMessage (String)*: Wallet load success or error message

ApiEvent.FILE_RECEIVED_EVENT: This event is used when a file message is received in the client app. And its data can be found in the **FileReceivedEvent** class.

Example:

```
        AppDataObserver.on().startObserver(ApiEvent.FILE_RECEIVED_EVENT,
event -> {
                   FileReceivedEvent fileReceivedEvent = (FileReceivedEvent)
        event;     });
```

FileReceivedEvent class has four properties

1. *fileMessageId (String)*: The messageId of the received file
2. *filePath (String):* The storage path where the received file is saving.
3. *sourceAddress (String)*: The file message sender node Id
4. *metaData (byte[]):* Any extra information regarding file messages like text message **1-byte** type. This information may contain file message type or other metadata information.

ApiEvent.FILE_PROGRESS_EVENT: This event is responsible for receiving file progress for both the sender and receiver side. The progress indicates how much file is sent and received in the sender and receiver side respectively. The progress data can be found in the **FileProgressEventClass**.

Example:

```
AppDataObserver.on().startObserver(ApiEvent.FILE_PROGRESS_EVENT, event ->
{
       FileProgressEvent fileProgressEvent = (FileProgressEvent) event;
});
```

The FileProgressEvent contains two properties.

1. *fileMessageId (String)*: The received file message-id.
2. *percentage (Int)*: The current progress of sent/received file

ApiEvent.FILE_TRANSFER_EVENT: This event is used when a file message sent/received successfully or any error occurs during sent/received a file. And this information can be found in the **FileTransferEvent** class.

Example:

```
     AppDataObserver.on().startObserver(ApiEvent.FILE_TRANSFER_EVENT,
event -> {
      FileTransferEvent fileTransferEvent = (FileTransferEvent) event;
}) ;
```

The **FileTransferEvent** two properties:

1. *fileMessageId (String):* The sent/received file messageId
2. *isSuccess (Boolean):* The status of the file message sent successfully or not the same as the file receive section. Any error occurs or not during file receiving.

ApiEvent.FILE_PENDING_EVENT: This API is used when the client app is not open but the service app is still running. And this message is actually pending for the client app. So all pending file message information can be found from this event and the data can be found in the **FilePendingEvent** class.

Example:

```
     AppDataObserver.on().startObserver(ApiEvent.FILE_PENDING_EVENT,
event -> {
           FilePendingEvent filePendingEvent = (FilePendingEvent) event;
});
```

FilePendingEvent class has below properties:

1. *senderId (String):* The file message sender (User) Id.
2. *contentId (String):* The file message Id
3. *contentPath (String):* The file message storage location
4. *progress (int):* The percentage of file message is received
5. *state (Int):* The Pending file message is failed or a success state
6. *isIncoming (boolean):* The file message is incoming or not
7. *contentMetaInfo (**ContentMetaInfo**):* This class holds some file meta info.
    a. *messgaeId (String):* FIle message id
    b. *messageType (Int)*: Client app can get its own file message type
    c. *fileType (Int):* Client app can get its file type. It is the custom file message separator. Like video message or audio message.

ApiEvent.SERVICE_DESTRYED: This event is used when the service app is destroyed completely. And the data can be found in the **ServiceDestroyed** class.

Example:

```
      AppDataObserver.on().startObserver(ApiEvent.SERVICE_DESTROYED, event
-> {
             ServiceDestroyed serviceDestroyed = (ServiceDestroyed) event;
});
```

The **ServiceDestroyed** class has only one property named *isFullDestroyed* that indicated the service app fully destroyed or not.

Deprecated **(Though this event is public but SDK take cares its associate action)**
ApiEvent.PERMISSION_INTERRUPTION: This event is a utility event. We know the client app is directly dependent on **Service** App and if any user forgets to accept/enable required permission in the **Service** app then the client app can notify by this event. And take associate action. The permission related can be found in the **PermissionInterruptionEvent** class.

Example:

```
      AppDataObserver.on().startObserver(ApiEvent.PERMISSION_INTERRUPTION,
event -> {
       PermissionInterruptionEvent permissionInterruptionEvent =
(PermissionInterruptionEvent) event;   });
```

The **PermissionInterruptionEvent** class has two properties**.**

1. *hardwareState (Int)*: This contains that if any hardware related permission is required or not like Bluetooth permission. The state value can be found in this **interface** `DataPlanConstants.INTERRUPTION_EVENT`

2. *permissions (List<String>)*: It contains all required permissions list.

ApiEvent.SERVICE_UPDATE: This event is used to show an alert that service update available if service app in the background. And its data contains a boolean value that indicates update avail available or not. The data can be found in the **ServiceUpdate** class.

Example:

```
        AppDataObserver.on().startObserver(ApiEvent.SERVICE_UPDATE, event ->
{
         ServiceUpdate serviceUpdate = (ServiceUpdate) event;   });
```

The **ServiceUpdate** class has only one property:

isNeeded (Boolean): True means service update available.

Note: In the future, this event will be private and the update alert will be shown from SDK.