

# Oracle® Banking APIs

## Aggregator Guide



Patchset Release 22.2.6.0.0

G28184-01

April 2025

ORACLE®

Copyright © 2006, 2025, Oracle and/or its affiliates.

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software, software documentation, data (as defined in the Federal Acquisition Regulation), or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, then the following notice is applicable:

U.S. GOVERNMENT END USERS: Oracle programs (including any operating system, integrated software, any programs embedded, installed, or activated on delivered hardware, and modifications of such programs) and Oracle computer documentation or other Oracle data delivered to or accessed by U.S. Government end users are "commercial computer software," "commercial computer software documentation," or "limited rights data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, reproduction, duplication, release, display, disclosure, modification, preparation of derivative works, and/or adaptation of i) Oracle programs (including any operating system, integrated software, any programs embedded, installed, or activated on delivered hardware, and modifications of such programs), ii) Oracle computer documentation and/or iii) other Oracle data, is subject to the rights and limitations specified in the license contained in the applicable contract. The terms governing the U.S. Government's use of Oracle cloud services are defined by the applicable contract for such services. No other rights are granted to the U.S. Government.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle®, Java, MySQL, and NetSuite are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Inside are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Epyc, and the AMD logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

This software or hardware and documentation may provide access to or information about content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services unless otherwise set forth in an applicable agreement between you and Oracle. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services, except as set forth in an applicable agreement between you and Oracle.

# Contents

## Preface

---

|                             |    |
|-----------------------------|----|
| Purpose                     | iv |
| Audience                    | iv |
| Documentation Accessibility | iv |
| Critical Patches            | iv |
| Diversity and Inclusion     | v  |
| Conventions                 | v  |
| Related Resources           | v  |
| Screenshot Disclaimer       | v  |
| Acronyms and Abbreviations  | v  |

## 1

## Aggregator Service

---

|       |  |     |
|-------|--|-----|
| 1.1   | Implementation                               | 1-1 |
| 1.2   | Implementation Details of Individual Service | 1-2 |
| 1.2.1 | SCF Implementation                           | 1-2 |
| 1.2.2 | VAM Implementation                           | 1-7 |
| 1.2.3 | Receivables Implementation                   | 1-9 |

## Index

---

# Preface

- [Purpose](#)
- [Audience](#)
- [Documentation Accessibility](#)
- [Critical Patches](#)
- [Diversity and Inclusion](#)
- [Conventions](#)
- [Related Resources](#)
- [Screenshot Disclaimer](#)
- [Acronyms and Abbreviations](#)

## Purpose

This guide is designed to help acquaint you with the Oracle Banking Digital Experience application. This guide provides answers to specific features and procedures that the user need to be aware of the module to function successfully.

## Audience

This document is intended for the following audience:

- Customers
- Partners

## Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at <http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc>.

### **Access to Oracle Support**

Oracle customer access to and use of Oracle support services will be pursuant to the terms and conditions specified in their Oracle order for the applicable services.

## Critical Patches

Oracle advises customers to get all their security vulnerability information from the Oracle Critical Patch Update Advisory, which is available at [Critical Patches, Security Alerts and](#)

**Bulletins.** All critical patches should be applied in a timely manner to ensure effective security, as strongly recommended by [Oracle Software Security Assurance](#).

## Diversity and Inclusion

Oracle is fully committed to diversity and inclusion. Oracle respects and values having a diverse workforce that increases thought leadership and innovation. As part of our initiative to build a more inclusive culture that positively impacts our employees, customers, and partners, we are working to remove insensitive terms from our products and documentation. We are also mindful of the necessity to maintain compatibility with our customers' existing technologies and the need to ensure continuity of service as Oracle's offerings and industry standards evolve. Because of these technical constraints, our effort to remove insensitive terms is ongoing and will take time and external cooperation.

## Conventions

The following text conventions are used in this document:

| Convention             | Meaning  |
|------------------------|--|
| <b>boldface</b>        | Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.         |
| <i>italic</i>          | Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.                          |
| <code>monospace</code> | Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter. |

## Related Resources

For more information on any related features, refer to the following documents:

- [Oracle Banking APIs Installation Manuals](#)

## Screenshot Disclaimer

Personal information used in the interface or documents is dummy and does not exist in the real world. It is only for reference purposes.

## Acronyms and Abbreviations

The list of the acronyms and abbreviations used in this guide are as follows:

**Table 1 Acronyms and Abbreviations**

| Abbreviation | Description         |
|--------------|---------------------|
| OBAPI        | Oracle Banking APIs |

# 1

## Aggregator Service

It is a generic service to list the required set of data like any normal List Service with the only difference that the data is in the most summarized view possible.

This service is mostly used to draw graphical widgets on the screen that represents the summary of a particular domain.

- [Implementation](#)  
Returns the aggregated data requested for the given resource like for instance Account, for given data parameter like for instance Closing Balance, for given interval like for instance Daily, Monthly, for given period.
- [Implementation Details of Individual Service](#)

### 1.1 Implementation

Returns the aggregated data requested for the given resource like for instance Account, for given data parameter like for instance Closing Balance, for given interval like for instance Daily, Monthly, for given period.

It has following parameters:

- resource - Resource for which the aggregation is required like Account. (Path Parameter)
- data - Data of the resource for which the aggregation is to be done like Closing balance for Account. (Query Parameter)
- grouping - Grouping for which the aggregation is required like type of transaction CREDIT or DEBIT. It is not mandatory to specify grouping. If not then default value will be "DEFAULT". (Query Parameter)
- interval - The interval at which the aggregated data is required like Daily, Monthly. It is not mandatory to specify interval. If not then default value will be "D – Daily". All available values are D - Daily, W - Weekly, F – Fortnightly, M - Monthly, Q - Quarterly, Y – Yearly. (Query Parameter)
- count - The number of times the aggregated data is required at given intervals. It is not mandatory to specify count. If not then default value will be 1. (Query Parameter)
- q - The generic filtering parameter. (Query Parameter)
- sortBy - sorting parameter to sort q param results. (Query Parameter)
- maxRecords - max records parameter to restrict count of q param results. (Query Parameter)

Now every combination of resource, data and grouping has its own service / implementation and an entry for the same is made in the seed.

Now every combination of resource, data and grouping has its own service / implementation and an entry for the same is made in the seed.

Now initially , the call goes to aggregator REST API with above mentioned parameters from which the control goes to aggregator service. Finally in aggregator service the fully qualified

name of the actual service / implementation is fetched from the DB based on the combination of “resource”, “data” and “grouping” and actual aggregation is done.

For every service / implementation, an entry is made in DIGX\_FW\_CONFIG\_ALL\_B where prop\_id is “resource.data.grouping” and category\_id is “AggregatorConfig”. Now for one implementation there can be multiple grouping possible. In that case the prop\_id will be “resource.data.grouping1, grouping2, ...groupingN”.

### **Example**

For every service / implementation, an entry is made in DIGX\_FW\_CONFIG\_ALL\_B where prop\_id is “resource.data.grouping” and category\_id is “AggregatorConfig”. Now for one implementation there can be multiple grouping possible. In that case the prop\_id will be “resource.data.grouping1, grouping2, ...groupingN”.

- **resource** - Invoice
- **data** - Amount
- **grouping** - Program

Thus, in this case the prop\_id will be “invoices.Amount.Program” where prop\_value contains the fully qualified name of the service which represents the implementation for the same.

Similarly, if multiple grouping is to be done like based on program and currency, then the prop\_id will be “invoice.Amount.Program,Currency” or “invoice.Amount.Currency,Program”.



#### **Note:**

In case of multiple grouping, the order of comma separated grouping names used in prop\_id and the ones sent in UI should be same.

**i.e. In the above case, If the prop\_id used is “invoice.Amount.Program,Currency” then from UI grouping should be sent in query parameter as “grouping=Program,Currency”.**

**\*\*\*\* In case of no grouping required then prop\_id will look like “resource.data.DEFAULT” i.e. “invoice.Amount.DEFAULT”**

Now, the parameters like q, sortBy and maxRecords are the parameters for QQuery implementation which are used similarly the way they are used in other OBAPI services.

In case of aggregator service , these parameters will be directly passed to the implementation service where they can be used in actual call made to get the original data before aggregation.

## 1.2 Implementation Details of Individual Service

- [SCF Implementation](#)
- [VAM Implementation](#)
- [Receivables Implementation](#)

### 1.2.1 SCF Implementation

#### **Case 1**

We want the list of top programs between the logged in party and specified associated party for both roles of logged in party - buyer and supplier. To achieve this we fire the list of Invoices



with below specified query parameters and then group them by program code (grouping invoices of same program) and finally calculate the sum of invoice amount for each group.

**Prop\_id:** invoices.Amount.Program

**URL:** /digx-scf/v1/aggregator/invoices

**Request Parameters:**

**data:** Amount

**grouping:** Program

**q:**

1. Role of the logged in party that is used to get list of invoices(B or S) .
2. Invoice status - ACCEPTED, RAISED, FINANCED, PARTIAL\_FINANCED to get outstanding invoices.
3. Payment status – UNPAID, PART\_PAID, OVERDUE to get outstanding invoices.
4. Associated party id – To get list of invoices only linked between logged in party id and associated party id specified.

**sortBy:** We want only top programs so we sort the groups by total amount. Thus we send amount and DESC(Descending).

**maxRecords:** We want only top 5 programs and thus we send 5.

**Response:**

```
{
  "aggregatedData": {
    "resource": "invoices",
    "groups": [{
      "id": "HPRF Program~HP Reverse Factoring~A~B",
      "intervals": [{
        "amount": {
          "currency": "GBP",
          "amount": 62739.98688000
        },
        "count": 4
      }]
    }, {
      "id": "HPPRGFIN1~HPPRGFIN1~A~B",
      "intervals": [
        {
          "amount": {
            "currency": "GBP",
            "amount": 8760.63785888
          },
          "count": 8
        }
      ]
    }
  ]
}
```

Here, “groups” is the array which contains list of data after grouping. i.e. different groups. Inside group, “id” represents the unique Id of that group. In the response of this API, “id” will

always be combination of program code, program name, relation of logged in party in that program (A or CP) and role of logged in party in that program (B or S), all tilde(~) separated.

Intervals contains the actual data (in this case, the total amount for each program) at various intervals. Here since no interval is specified there will always be one element in intervals array.

## Case 2

We want the list of programs currency wise between the logged in party and specified associated party for both roles of logged in party - buyer and supplier. To achieve this we fire the list of Invoices with below specified query parameters and then group them by program code (grouping invoices of same program), then group them by currency(grouping invoices of same currency for every program) and finally calculate the sum of invoice amount for each group.

**Prop\_id:** invoices.Amount.Currency,Program or invoices.Amount.Program,Currency

**URL:** /digx-scf/v1/aggregator/invoices

### Request Parameters:

**data:** Amount

**grouping:** Currency,Program or Program,Currency

**q:**

1. Role of the logged in party that is used to get list of invoices(B or S) .
2. Invoice status - ACCEPTED, RAISED, FINANCED, PARTIAL\_FINANCED to get outstanding invoices.
3. Payment status – UNPAID, PART\_PAID, OVERDUE to get outstanding invoices.
4. Associated party id – To get list of invoices only linked between logged in party id and associated party id specified.

### Response:

```
{
  "aggregatedData": {
    "resource": "invoices",
    "groups": [{
      "identifiers": ["GBP", "HPFactoringWRec~HP Factoring WRec~A~S"],
      "intervals": [{
        "amount": {
          "currency": "GBP",
          "amount": 7426.00
        },
        "count": 37
      }]
    },
    { "identifiers": ["USD", "HPFactoringWRec~HP Factoring WRec~A~S"],
      "intervals": [
        { "amount": {
          "currency": "USD",
          "amount": 39840.00
        },
        "count": 295
        }
      ]
    }
  ]
}
```

```
    }
  }
```

Here, “groups” is the array which contains list of data after grouping. i.e. different groups. Inside group, “identifiers” represents the unique Id of that group. Since, multiple grouping is done, id is the list containing 2 elements. First is Currency and second is the combination of program code, program name, relation of logged in party in that program (A or CP) and role of logged in party in that program (B or S), all tilde(~) separated.

Intervals contains the actual data (in this case, the total amount for each program, currency wise) at various intervals. Here since no interval is specified there will always be one element in intervals array.

### **Case 3**

We want the list of top associated parties linked with logged in party in a particular program. To achieve this we fire the list of Invoices with below specified query parameters and then group them by associated party Id (grouping invoices of same associated party) and finally calculate the sum of invoice amount for each group.

**Prop\_id:** invoices.Amount.AssociatedParty

**URL:** /digx-scf/v1/aggregator/invoices

#### **Request Parameters:**

**data:** Amount

**grouping:** AssociatedParty

**q:**

1. Role of the logged in party that is used to get list of invoices(B or S) .
2. Invoice status - ACCEPTED, RAISED, FINANCED, PARTIAL\_FINANCED to get outstanding invoices.
3. Payment status – UNPAID, PART\_PAID, OVERDUE to get outstanding invoices.
4. program code – To get list of invoices only linked between logged in party id and associated party id in the specified program.

**sortBy:** We want only top associated parties so we sort the groups by total amount. Thus we send amount and DESC(Descending).

**maxRecords:** We want only top 10 associated parties and thus we send 10.

#### **Response:**

```
{
  "aggregatedData": {
    "resource": "invoices",
    "groups": [{
      "id": " E4228ED58341003545623EDC7319024990E5C38ACB60 ~***728 ~TURBO
TEXTILES ",
      "intervals": [{
        "amount": {
          "currency": "GBP",
          "amount": 7426.00 },
        "count": 37
      ]
    }
  ]
}
```

```

    ]]
  },
  { "id": " E4228ED58341003545623EDC7319024990E5C38ACB90~*****C001
~TestCP02 ",
    "intervals":
      [{ "amount":{
          "currency":"USD",
          "amount":39840.00 },
          "count":295
        }]
      ]
  }
}

```

Here, “groups” is the array which contains list of data after grouping. i.e. different groups. Inside group, “id” represents the unique Id of that group. In the response of this API, “id” will always be combination of associated party id (hashed value), associated party id (display value), all tilde(~) separated.

Intervals contains the actual data (in this case, the total amount for each associated party) at various intervals. Here since no interval is specified there will always be one element in intervals array.

#### **Case 4**

We want the list of associated parties currency wise linked with logged in party in a particular program. To achieve this we fire the list of with below specified query parameters and then group them by associated party id (grouping invoices of same associated party), then group them by currency(grouping invoices of same currency for every associated party) and finally calculate the sum of invoice amount for each group.

**Prop\_id:** invoices.Amount.AssociatedParty,Currency or  
invoices.Amount.Currency,AssociatedParty

**URL:**/digx-scf/v1/aggregator/invoices

#### **Request Parameters:**

**data:** Amount

**grouping:** AssociatedParty,Currency or Currency,AssociatedParty

**q:**

1. Role of the logged in party that is used to get list of invoices(B or S) .
2. Invoice status - ACCEPTED, RAISED, FINANCED, PARTIAL\_FINANCED to get outstanding invoices.
3. Payment status – UNPAID, PART\_PAID, OVERDUE to get outstanding invoices.
4. program code – To get list of invoices only linked between logged in party id and associated party id in the specified program.

#### **Response:**

```

{
  "aggregatedData": {
    "resource": "invoices",
    "groups":

```

```

      [{"identifiers":
        ["GBP", " E4228ED58341003545623EDC7319024990E5C38ACB60 ~***728
~TURBO TEXTILES "],
        "intervals": [{ "amount":
          {"currency":"GBP",
            "amount":7426.00 },
            "count":37
          }]
        },
        {"identifiers": ["USD", "
E4228ED58341003545623EDC7319024990E5C38ACB60 ~***728 ~TURBO TEXTILES "],
        "intervals":
          [{ "amount":{
            "currency":"USD",
            "amount":39840.00 },
            "count":295
          }]
        }
      ]
    }
  }
}

```

Here, “groups” is the array which contains list of data after grouping. i.e. different groups. Inside group, “identifiers” represents the unique Id of that group. Since, multiple grouping is done, id is the list containing 2 elements. First is Currency and second is the combination of associated party id (hashed value), associated party id (display value), all tilde(~) separated.

Intervals contains the actual data (in this case, the total amount for each associated party, currency wise) at various intervals. Here since no interval is specified there will always be one element in intervals array.

## 1.2.2 VAM Implementation

### Case 1

Fetching list of value dated balances for the top N virtual accounts with respect to available balance for a selected virtual entity and currency. It fetched the

**Prop\_id:** virtualAccounts.valueDated.DEFAULT

**URL:** GET + /digx-vam/v1/aggregator/resource/virtualAccounts

#### **Request Parameters:**

**data:** valueDated

**maxRecords:** 5 (Integer for number of virtual accounts)

**q:**

- virtualEntityId - the selected virtual entity id filter on virtual accounts
- vStatus – only open virtual accounts to be fetched
- availableBalance.currency – the selected currency filter on virtual accounts

**sortParams:**

- sortBy : availableBalance.amount

- sortOrder: DESC (Fetches top N)

**Response:**

```
{
  "aggregatedData": {
    "resource": "virtualAccounts",
    "groups": [{
      "id": {
        "displayValue": "xxxxxxxxxxxx0096",
        "value":
"C56C880F40EA1F354870342328EED1323799A835BE1813AA"
      },
      "intervals": [{
        "amount": {
          "currency": "GBP",
          "amount": -165
        },
        "date": "2018-10-02T00:00:00"
      }
    ]
  }
}
```

Id - is the virtual account number each group in groups array represents the balance for the value date in the group.

**Case 2**

Fetching list of virtual accounts for a selected virtual entity and group the virtual accounts based on the currency and aggregate the availableBalance of the virtual account to provide currency wise distribution to the user.

**Prop\_id:** virtualAccounts.availableBalance.DEFAULT

**URL:** GET + /digx-vam/v1/aggregator/resource/virtualAccounts

**Request Parameters:**

**data:** availableBalance

**grouping:** Currency,Program or Program,Currency

**q:**

- virtualEntityId - the selected virtual entity id filter on virtual accounts
- vStatus – only open virtual accounts to be fetched

**sortParams:**

- sortBy : availableBalance.amount
- sortOrder: DESC (Fetches top N)

**Response:**

```
{
  "aggregatedData": {
    "resource": "virtualAccounts",
```

```

        "groups": [{
            "id": "EUR",
            "intervals": [{
                "amount": {
                    "currency": "EUR",
                    "amount": 1329
                },
                "count": 2
            }]
        }]
    }
}

```

Each group in the groups array represents the currency and its sum for the virtual accounts satisfying the criteria in the request and the number of virtual accounts in that criteria.

## 1.2.3 Receivables Implementation

### 2.2.3.1 Purchase Order

#### Case 1

We want the list of Purchase Orders status wise linked with logged in party. To achieve this we fire the list of purchase orders with below specified query parameters and then group them by status (grouping purchase orders of same status) and finally calculate the sum of purchase order amount for each group.

**Prop\_id:** purchaseorders.Amount.Status

**URL:** /digx-scf/v1/aggregator/purchaseorders

**Request Parameters:**

**data:** Amount

**grouping:** Status

**q:1.** Role of the logged in party that is used to get list of purchase orders(B or S) .

**Response:**

```

{ "aggregatedData":
  {
    "resource": "purchaseorders", "groups":
    [{
      "identifiers": ["ACCEPTED"],
      "intervals": [{
        "amount": {
          "currency":
"EUR",
"amount":132708027.4107700091004397884741905500050052069127559661865234375
        },
        "count": 70
      }]
    },
    {

```

```

        "identifiers": ["RAISED"],
        "intervals": [{
            "amount": {
                "currency":
"EUR",
                "amount":
4071865.5212500003207022947204762886030948720872402191162109375
            },
            "count": 61
        }],
        {
            "identifiers": ["REJECTED"], "intervals":
[
            {
                "amount": {
                    "currency":
"EUR",
                    "amount":
20205.57000000000016314682937945690355263650417327880859375
                },
                "count": 7
            }
        ],
        {
            "identifiers": ["CANCELLED"],
            "intervals": [{
                "amount": {
                    "currency":
"EUR",
                    "amount":1653850.0000000000133537625401913828682154417037963867187500
                },
                "count": 4
            }
        ]
    }
}

```

Here, “groups” is the array which contains list of data after grouping. i.e. different groups. Inside group, “identifiers” represents the unique Id of that group. Since, grouping is done on the basis of purchase order status, id contains purchase order’s status.

Intervals contains the actual data (in this case, the total amount for each status wise) at various intervals. Here since no interval is specified there will always be one element in intervals array.

## **Case 2**

We want the list of Top 10 associated parties purchase order status wise linked with logged in party. To achieve this we fire the list of purchase orders with below specified query parameters and then group them by associated party id (grouping purchase orders of same associated party), then group them by purchase order status(grouping purchase orders of same status for every associated party) and finally calculate the sum of purchase order amount for each group.

**Prop\_id:** purchaseorders.Amount.AssociatedParty,Status

**URL:** /digx-scf/v1/aggregator/purchaseorders



**Request Parameters:****data:** Amount**grouping:** AssociatedParty,Status**q:**

- Role of the logged in party that is used to get list of purchase orders(B or S) .
- Invoice status - ACCEPTED, RAISED.

**Response:**

```
{  "aggregatedData":
  {
    "resource": "purchaseorders", "groups":
    [{
      "identifiers":

["98DCBD13A0F3EA4F5EDE19325B4CD2D30A1C949B838D0E171B1B~***000153~LINKINVA23Dec
",
      "ACCEPTED"],
      "intervals": [{
        "amount": {
          "currency":

"EUR",
          "amount":

109765554.8000000008862854272706499614287167787551879882812500

        },
        "count": 1
      }
    ]
  },
  {
    "identifiers":

["E6AFBD17A6F1717F4C5BCEE75406C9BB37C8DF99577A~***462~ABZ
Solutions", "ACCEPTED"],
    "intervals": [{
      "amount": {
        "currency":

"EUR",
        "amount":

22904335.7207700002345062100683747985385707579553127288818359375

      },
      "count": 62
    }
  ]
},
{
  "identifiers":

["E6AFBD17A6F1717F4C5BCEE75406C9BB37C8DF99577A~***462~ABZ Solutions",
"RAISED"],
  "intervals": [{
    "amount": {
      "currency":

"EUR",
```

```

"amount":4066573.1572500003202749707664764855508110485970973968505859375

    },
    "count": 45
  }]
  {
    "identifiers":
["98DCBD13A0F3EB4D5ADBFB7E68621176D18D733CA71A0B515B69~***000077~Septoneparty"
,
    "ACCEPTED"],
    "intervals": [{
      "amount": {
        "currency":
"EUR",

"amount":22000.0000000000001776356839400250464677810668945312500000

    },
    "count": 1
  }]
  {
    "identifiers":
["E6AFBC14A1F57BA066463EE6C10B5E5D5324274D7899~***716~AugBuyer",
    "ACCEPTED"],
    "intervals": [{
      "amount": {
        "currency":
"EUR",

"amount":10636.890000000000008588596500658240984193980693817138671875

    },
    "count": 5    }]
    {
      "identifiers":

["98DCBD13A0F3EB4C5BD1361685D7E30CC39EA19106076A3CB8DB~***000066~LinkInvBuyCp"
,
      "ACCEPTED"],
      "intervals": [{
        "amount": {
          "currency":
"EUR",
          "amount":
5500.0000000000000444089209850062616169452667236328125000

    },
    "count": 1
  }]
  {
    "identifiers":
["E6AFBC14A1F57BA066463EE6C10B5E5D5324274D7899~***716~AugBuyer",
    "RAISED"],
    "intervals": [{
      "amount": {
        "currency":

```

```

"EUR",
  "amount":
5292.364000000000042732395399980305228382349014282226562500
},
"count": 16
}]
}}

```

Here, “groups” is the array which contains list of data after grouping. i.e. different groups. Inside group, “identifiers” represents the unique Id of that group. Since, multiple grouping is done, id is the list containing 2 elements. First is the combination of associated party id (hashed value), associated party id (display value) and associated party name, all tilde(~) separated and second is the purchase order status.

Intervals contains the actual data (in this case, the total amount for each associated party, status wise) at various intervals. Here since no interval is specified there will always be one element in intervals array.

### 2.2.3.2 Reconciliation

#### Case 1

We want the list of unmatched payments currency wise for the logged in party. To achieve this we fire the list of payments with below specified query parameters and then group them by currency (grouping payments of same currency), then group them by payment type (grouping payments of same type for every currency) and finally calculate the sum of payment amount for each group.

**Prop\_id:** payments.Amount.Currency,Type

**URL:** /digx-cms/v1/aggregator/payments

**Request Parameters:**

**data:** Amount

**grouping:** Currency,Type

**q:1.**Payment Status - UNMATCHED .

**Response:**

```

{ "aggregatedData":
  {
    "resource": "payments", "groups":
    [{
      "identifiers": ["GBP", "C"],
      "intervals": [{
        "amount": {
          "currency":
"GBP",
        "amount": 835619
        },
        "count": 7
      }],
    },
  ]
}

```

```

        "identifiers": ["USD", "D"],
        "intervals": [{
          "amount": {
            "currency":
"USD",
            "amount": 346103
          },
          "count": 8
        }],
        {
          "identifiers": ["GBP", "D"],
          "intervals": [{
            "amount": {
              "currency":
"GBP",
              "amount": 333903
            },
            "count": 7
          }],
          {
            "identifiers": ["USD", "C"],
            "intervals": [{
              "amount": {
                "currency":
"USD",
                "amount": 40200
              },
              "count": 5
            }],
            }
          }
        }
  
```

Here, “groups” is the array which contains list of data after grouping. i.e. different groups. Inside group, “identifiers” represents the unique Id of that group. Since, multiple grouping is done, id is the list containing 2 elements. First is the currency and second is the payment type.

Intervals contains the actual data (in this case, the total amount for each currency, payment type wise) at various intervals. Here since no interval is specified there will always be one element in intervals array.

## **Case 2**

We want the list of unreconciled invoices currency wise for the logged in party. To achieve this we fire the list of invoices with below specified query parameters and then group them by currency (grouping invoices of same currency), then group them by role wise (grouping invoices of same role for every currency) and finally calculate the sum of invoice amount for each group.

**Prop\_id:** invoices.Amount.Currency,Role

**URL:** /digx-invoice/v1/aggregator/invoices

**Request Parameters:**

**data:** Amount

**grouping:** Currency,Role

**q:1.Payment Status - UNPAID .****Response:**

```
{ "aggregatedData":
  {
    "resource": "invoices", "groups":
    [{
      "identifiers": ["GBP", "B"],
      "intervals": [{
        "amount": {
          "currency":
"GBP",
          "amount": 1047165452.0
        },
        "count": 281
      }]
    },
    {
      "identifiers": ["GBP", "S"],
      "intervals": [{
        "amount": {
          "currency":
"GBP",
          "amount": 4742388.0
        },
        "count": 75
      }]
    },
    {
      "identifiers": ["USD", "B"],
      "intervals": [{
        "amount": {
          "currency":
"USD",
          "amount": 17900.0
        },
        "count": 5
      }]
    },
    {
      "identifiers": ["USD", "S"],
      "intervals": [{
        "amount": {
          "currency":
"USD",
          "amount": 5000.0
        },
        "count": 3
      }]
    },
    {
      "identifiers": ["INR", "S"],
      "intervals": [{
        "amount": {
          "currency":
"INR",
          "amount": 5000.0
        },
        "count": 1
      }]
    }
  ]
}
```

```

    }],
    {
      "identifiers": ["LAK", "B"],
      "intervals": [{
        "amount": {
          "currency":
"LAK",
          "amount": 4401
        },
        "count": 37
      }]
    },
    {
      "identifiers": ["LAK", "S"],
      "intervals": [{
        "amount": {
          "currency":
"LAK",
          "amount": 2024
        },
        "count": 7
      }]
    }
  ]
}
```

Here, “groups” is the array which contains list of data after grouping. i.e. different groups. Inside group, “identifiers” represents the unique Id of that group. Since, multiple grouping is done, id is the list containing 2 elements. First is the currency and second is the associated party role.

Intervals contains the actual data (in this case, the total amount for each currency, associated party role wise) at various intervals. Here since no interval is specified there will always be one element in intervals array.

### **Case 3**

We want the list of unreconciled cashflows currency wise for the logged in party. To achieve this we fire the list of cashflows with below specified query parameters and then group them by currency (grouping cashflows of same currency), then group them by cashflow type(grouping cashflows of same type for every currency) and finally calculate the sum of cashflow amount for each group.

**Prop\_id:** cashflows.Amount.Currency,Type

**URL:** /digx-cms/v1/aggregator/cashflows

**Request Parameters:**

**data:** Amount

**grouping:** Currency,Type

**q:1.Reconciliation Status - UNRECONCILED .**

**Response:**

```

{ "aggregatedData":
  {
    "resource": "cashflows", "groups":
    [{
```

```

        "identifiers": ["LAK", "I"],
        "intervals": [{ "amount":
{
    "currency":
"LAK",
    "amount": 573993369
    },
    "count": 82
    }]
    {
        "identifiers": ["GBP", "I"],
        "intervals": [{
            "amount": {
            "currency":
"GBP",
            "amount": 59712749
            },
            "count": 194
            }]
            {
                "identifiers": ["USD", "I"],
                "intervals": [{
                    "amount": {
                    "currency":
"USD",
                    "amount": 36663338
                    },
                    "count": 177
                    }]
                    {
                        "identifiers": ["GBP", "O"],
                        "intervals": [{
                            "amount": {
                            "currency":
"GBP",
                            "amount": 1666225
                            },
                            "count": 67
                            }]
                            {
                                "identifiers": ["LBP", "I"],
                                "intervals": [{
                                    "amount": {
                                    "currency":
"LBP",
                                    "amount": 991545
                                    },
                                    "count": 4 }]
                                    },
                                    {
                                        "identifiers": ["USD", "O"],
                                        "intervals": [{
                                            "amount": {
                                            "currency":
"USD",
                                            "amount": 51044.33

```

```

    },
    "count": 264
  }],
  {
    "identifiers": ["EUR", "I"],
    "intervals": [{
      "amount": {
        "currency":
"EUR",
        "amount": 28547.17
      },
      "count": 34
    }],
    "identifiers": ["LAK", "O"],
    "intervals": [{
      "amount": {
        "currency":
"LAK",
        "amount": 4567
      },
      "count": 2
    }],
    "count": 2
  }],
  "count": 2
}]]

```

Here, “groups” is the array which contains list of data after grouping. i.e. different groups. Inside group, “identifiers” represents the unique Id of that group. Since, multiple grouping is done, id is the list containing 2 elements. First is the currency and second is the cashflow type.

Intervals contains the actual data (in this case, the total amount for each currency, cashflow type wise) at various intervals. Here since no interval is specified there will always be one element in intervals array.

#### **Case 4**

We want the list of payments status wise for the logged in party. To achieve this we fire the list of payments with below specified query parameters and then group them by status (grouping payments of same status), then group them by type (grouping payments of same type for every status) and finally calculate the sum of payment amount for each group.

**Prop\_id:** payments.Amount.Status,Type

**URL:** /digx-cms/v1/aggregator/payments

**Request Parameters:**

**data:** Amount

**grouping:** Status,Type

**q:1.**Payment Date – from date and to date .

**Response:**

```

{ "aggregatedData":
  {
    "resource": "payments", "groups":
    [{

```



```
        "identifiers": ["MATCHED", "C"],
        "intervals": [{
          "amount": {
            "currency":
"USD",
            "amount": 291300
          },
          "count": 57
        }]
      },
      {
        "identifiers": ["PART_MATCHED",
"C"],
        "intervals": [{
          "amount": {
            "currency":
"GBP",
            "amount": 9700
          },
          "count": 1
        }]
      },
      {
        "identifiers": ["PART_MATCHED",
"D"],
        "intervals": [{
          "amount": {
            "currency":
"GBP",
            "amount": 32900
          },
          "count": 6
        }]
      },
      {
        "identifiers": ["RECON_NA", "D"],
        "intervals": [{
          "amount": {
            "currency":
"USD",
            "amount": 5200
          },
          "count": 1
        }]
      },
      {
        "identifiers": ["MATCHED", "D"],
        "intervals": [{
          "amount": {
            "currency":
"GBP",
            "amount": 580900
          },
          "count": 41
        }]
      }
    ]
  }
}
```

Here, “groups” is the array which contains list of data after grouping. i.e. different groups. Inside group, “identifiers” represents the unique Id of that group. Since, multiple grouping is

done, id is the list containing 2 elements. First is the payment status and second is the payment type.

Intervals contains the actual data (in this case, the total amount for each status, payment type wise) at various intervals. Here since no interval is specified there will always be one element in intervals array.

### **Case 5**

We want the list of unmatched payments entity wise for the logged in party. To achieve this we fire the list of payments with below specified query parameters and then group them by payment entity (grouping payments of same entity), then group them by payment type(grouping payments of same type for every entity) and finally calculate the sum of payment amount for each group.

**Prop\_id:** payments.Amount.Entity,Type

**URL:** /digx-cms/v1/aggregator/payments

**Request Parameters:**

**data:** Amount

**grouping:** Entity,Type

**q:1.**Payment Status - UNMATCHED .

**Response:**

```
{  "aggregatedData":
  {  "resource":
    "payments",  "groups":
    [{
      "identifiers": ["E", "C"],
      "intervals": [{
        "amount": {
          "currency":
"GBP",
          "amount": 140412
        },
        "count": 5
      }
    }, {
      "identifiers": ["I", "C"],
      "intervals": [{
        "amount": {
          "currency":
"USD",
          "amount": 735407
        },
        "count": 7
      }
    }, {
      "identifiers": ["I", "D"],
      "intervals": [{
        "amount": {
          "currency":
"USD",
          "amount": 553005
        },
        "count": 9
      }
    }
  ]
}
```

```

    }
    "identifiers": ["E", "D"],
    "intervals": [{
      "amount": {
        "currency":
"GBP",
        "amount": 127001
      },
      "count": 6
    }]
  }
}

```

Here, “groups” is the array which contains list of data after grouping. i.e. different groups. Inside group, “identifiers” represents the unique Id of that group. Since, multiple grouping is done, id is the list containing 2 elements. First is the payment entity and second is the payment type.

Intervals contains the actual data (in this case, the total amount for each entity, payment type wise) at various intervals. Here since no interval is specified there will always be one element in intervals array.

### **Case 6**

We want the list of payments allocation status wise for the logged in party. To achieve this we fire the list of payments with below specified query parameters and then group them by allocation status (grouping payments of same allocation status), then group them by payment type wise (grouping payments of same type for every allocation status) .

**Prop\_id:** payments.Count.AllocationStatus,Type

**URL:** /digx-cms/v1/aggregator/payments

**Request Parameters:**

**data:** Count

**grouping:** AllocationStatus,Type

**q:1.**Payment Date – from date and to date .

**Response:**

```

{  "aggregatedData":
  {
    "resource": "payments",  "groups":
    [{
      "identifiers": ["FAILED", "C"],
      "intervals": [{
        "count": 3
      }]
    },
    {
      "identifiers": ["UNALLOCATED",
"C"],
      "intervals": [{
        "count": 55
      }]
    },
    {

```

```

        "identifiers": ["UNALLOCATED",
"D"],
        "intervals": [{
            "count": 47
        }],
        {
            "identifiers": ["FAILED", "D"],
            "intervals": [{
                "count": 1
            }]
        }
    ]
}

```

Here, “groups” is the array which contains list of data after grouping. i.e. different groups. Inside group, “identifiers” represents the unique Id of that group. Since, multiple grouping is done, id is the list containing 2 elements. First is the allocation status and second is the payment type.

Intervals contains the actual data (in this case, the total count for each allocation status, payment type wise) at various intervals. Here since no interval is specified there will always be one element in intervals array.

### **Case 7**

We want the list of allocated transactions allocation type wise for the logged in party. To achieve this we fire the list of payments with below specified query parameters and then group them by allocation type (grouping payments of same allocation type), then group them by payment type (grouping payments of same type for every allocation type) .

**Prop\_id:** allocations.Count.PaymentType,Type

**URL:** /digx-cms/v1/aggregator/allocations

**Request Parameters:**

**data:** Count

**grouping:** PaymentType,Type

**Response:**

```

{ "aggregatedData":
  { "resource":
    "allocations",    "groups":
    [{
      "identifiers": ["C", "M"],
      "intervals": [{
        "count": 102
      }],
      }, {
      "identifiers": ["D", "M"],
      "intervals": [{
        "count": 50
      }]
    }
  ]
}

```

Here, “groups” is the array which contains list of data after grouping. i.e. different groups. Inside group, “identifiers” represents the unique Id of that group. Since, multiple grouping is done, id is the list containing 2 elements. First is the payment type and second is the allocation type.

Intervals contains the actual data (in this case, the total count for each payment type, allocation type wise) at various intervals. Here since no interval is specified there will always be one element in intervals array.

# Index

## A

---

Aggregator Service, [1-1](#)

## I

---

Implementation, [1-1](#)

## R

---

Receivables Implementation, [1-9](#)

## S

---

SCF Implementation, [1-2](#)

## V

---

VAM Implementation, [1-7](#)