UNBXD
iOS SDK

Introduction

UnbxdSDK implements support for making network calls to Unbxd platform and lets to easily configure and integrate Unbxd Site Search in your eCommerce application.

The following features are currently supported with UnbxdSDK,

1. **Site search :** The Search API lets you interact with the Unbxd platform and implement all search related functionality with ease.
2. **Auto suggest :** Unbxd supports autocompletion of search queries and showcasing products relevant to query as they type.
3. **Browse** : The Browse API lets you interact with the Unbxd platform and implement all category related functionality with ease.
4. **Analytics** : SDK provides API for visitor events such as roduct clicks, products added to cart, orders, etc.
5. **Recommendations** : Unbxd supports product recommendations with help of recorded events.

Prerequisites

Before you get started with the integration, you would need to:

- Set up your Site Search Dashboard and obtain API key, Site key.
- Upload your Product Feed.

Supported Platform

UnbxdSDK is a dynamic framework programmed using swift 5. This can be integrated with iOS application with version 10.0 and above.

Installation

Cocoapods is a dependency manager for Cocoa projects. You can install it with below command:

```
$ gem install cocoa pods
```

To integrate UnbxdSDK into your Xcode project using Cocoapods, specify it in your Podfile:

```
source 'https://github.com/unbxd/iOS-SDK-Pod.git'
platform : ios, '10.0'
use_frameworks!

target '<your target name>' do
        pod 'Unbxd'
end
```

Run the below command:

```
$ pod install
```

Initialising SDK

- Import Unbxd framework as shown below:

```
import Unbxd
```

- Unbxd is initialised with API key and Site key.

```
let client = Client(siteKey: "<SITE-KEY>", apiKey: "<API-KEY>", logsConfig: LogsConfig)
```

- LogsConfig – Used to configure log level and provide folder path where log file to be saved.

  Eg: let logsFolderPath = "\(NSHomeDirectory())/Documents"
  LogsConfig(logLevel: .verbose, folderPath: logsFolderPath)

All the SDK methods are invoked on the instance of Client.

Using Search Methods

Search methods performs search with key and few other parameters. Search key is the mandatory argument and rest is applied for different criteria.

Query method signature:

*init(**key:** String, **rows**: Int? = nil, **start**: Int? = nil, **format**: ResponseFormat = .JSON, **spellCheck**: Bool = false, **analytics**: Bool = true, **statsField**: String? = nil, **variant**: Variant? = nil, **fields**:Array<String>? = nil, **facet**: Facet? = nil, **filter**: FilterAbstract? = nil, **categoryFilter**: CategoryFilterAbstract? = nil, **multipleFilter**: MultipleFilterAbstract? = nil, **fieldsSortOrder**: Array<FieldSortOrder>? = nil, **personalization**: Bool? = nill)*

Search method signature:

```
func search(query:SearchQuery, completion: @escaping (_ response:  Dictionary<String, Any>?,_ httpResponse: HTTPURLResponse?, _ error:Error?) -> Void) {
    // Handle response or request
}
```

Lets see how these arguments can be composed and passed in search() method invocation.

A. SEARCHQUERY

SearchQuery consists of searchKey parameter with other parameters.

Invoking Search method with key

```
let query = SearchQuery(key: "Shirt")
client.search(query: query, completion:{  (response, httpResponse, err) -> Void in
        //Handle response
})
```

B. FORMAT

The format parameter specifies the format of the response. Possible values are ‘json’ or ‘xml’. It is an optional parameter and the default value is ‘json’.

```
let query = SearchQuery(key: "Shirt", format: .XML)
client.search(query: query, completion:{(response, httpResponse, err)-> Void in
        //Handle response
})
```

C. START

The start parameter is used to offset the results by a specific number. It indicates offset in the complete result set of the products. It is an optional parameter and the default value is 0.

```
let query = SearchQuery(key: "Shirt", start: 2, format: .JSON)
client.search(query: query, completion: {(response, httpResponse, err)-> Void in
        //Handle response
})
```

D. ROWS

The rows parameter is used to paginate the results of a query. It indicates number of products in a single page. It is an optional parameter and the default value is 10, maximum value is 100.

```
let query = SearchQuery(key: "Shirt", rows: 20)
client.search(query: query, completion: {(response, httpResponse, err)-> Void in
        //Handle response
})
```

E. SPELLCHECK

The spellcheck feature provides spelling suggestions or spell-check for misspelled search queries.

```
let query = SearchQuery(key: "Shirt", spellCheck: true)
client.search(query: query, completion: {(response, httpResponse, err)-> Void in
        //Handle response
```

```
})
```

F. ANALYTICS

The analytics parameter enables or disables tracking the query hit for analytics. By default, tracking is enabled.

```
let query = SearchQuery(key: "Shirt", analytics: false)
client.search(query: query, completion: {(response, httpResponse, err)-> Void in
    //Handle response
})
```

G. STATS

The stats parameter gives information about the products with highest and lowest field value.

```
let query = SearchQuery(key: "Shirt", statsField: "price")
client.search(query: query, completion: {(response, httpResponse, err)-> Void in
    //Handle response
})
```

H. VARIANTS

Variants parameter enables or disables variants in the API response. It can take two values: “true” or “false”. Default value is “false”.

```
let query = SearchQuery(key: "Shirt", variant: Variant(has: true, count: 2))
client.search(query: query, completion: {(response, httpResponse, err)-> Void in
    //Handle response
})
```

If you want to get more than one variants in the API response, you can use ‘variantCount’ parameter. It can have any numerical value (eg, 1,2,3,etc) or “.max” (to get all the variants).

I. FIELDS

The fields parameter is used to specify the set of fields to be returned. When returning the results, only fields in the list will be included.

```
let query = SearchQuery(key: "Shirt", fields: ["title","vPrice"])
client.search(query: query, completion: {(response, httpResponse, err)-> Void in
        //Handle response
})
```

J. FACETS

Facets are the filters in the UI that allow visitors to narrow down result set based on product fields. Facet option can have 3 values:

1.**Multi-level**: The facer multilevel parameter is used to enable multi-level facets in the API response.

```
let query = SearchQuery(key: "Shirt", facet: .MultiLevel)
client.search(query: query, completion: {(response, httpResponse, err)-> Void in
    //Handle response
 })
```

2.**Multi-select**: This feature enables or disables the option to select multiple values within a facet or across facets for visitors.

```
let query = SearchQuery(key: "Shirt", facet: .MultiSelect)
client.search(query: query, completion: {(response, httpResponse, err)-> Void in
        //Handle response
})
```

3.**Selected**: The selected facet parameter enables or disables the Selected Facets in the API response.

Selected facet with field id and value id:

```
let query = SearchQuery(key: "Shirt", facet: .Selected(IdFilter(field: "76678", value:
"5001")))
client.search(query: query, completion: {(response, httpResponse, err)-> Void in
```

```
        //Handle response
    })
```

Selected facet with field name and value name:

```
    let query = SearchQuery(key: “Shirt", facet: .Selected(NameFilter(field: "Brand_uFilter", value: "Vince Camuto")))
    client.search(query: query, completion: {(response, httpResponse, err)-> Void in
        //Handle response
    })
```

K. FILTERING

Filtering can be performed on fields using field Id or field Name.

Three types of filters are supported:

1. Text

The text filter is used to filter products based on fields with string values such as color, gender, brand, etc. It can be defined in the API call in two ways:

- Using Field Ids

  IdFilter can be formed with 2 parameters.

  field: The id of the field on which the text filter is applied.

  value: The id of the value on which the results are filtered.

Example:

```
    let query = SearchQuery(key: "Shirt", filter: IdFilter(field: "76678", value: "5001"))
    client.search(query: query, completion: {(response, httpResponse, err)-> Void in
        //Handle response
    })
```

- Using Field Names

  Again NameFilter can be formed with 2 parameters.

  type: The id of the field on which the text filter is applied.

  value: The id of the value on which the results are filtered.

Example:

```
let query = SearchQuery(key: "Shirt", filter: NameFilter(field:  "vColor_uFilter",
value: "Black"))
 client.search(query: query, completion: {(response, httpResponse, err)-> Void in
        //Handle response
 })
```

2. Range

The range filter is used to filter products based on fields with dataypes - *date*, *number* or *decimal*. It can be defined in the API in two ways:

- Using Field Id

  Filter Range of type id is built using IdFilterRange class and it can be initialised with below parameters.

  field: The id of the field on which the text filter is applied.
  lower: The id of the lower limit of the range.
  upper: The id of the upper limit of the range.

  Example:

```
let query = SearchQuery(key: "Shirt", filter: IdFilterRange(field:"76678", lower:
"2034", upper: "8906"))
client.search(query: query, completion: {(response, httpResponse, err)-> Void in
        //Handle response
})
```

- Using Field Name

  Filter Range of type name is built using NameFilterRange class and it can be initialised with below parameters.

  field: The name of the field on which the text filter is applied.
  lower: The name of the lower limit of the range.
  upper: The name of the upper limit of the range.

Example:

```
let query = SearchQuery(key: "Shirt", filter: NameFilterRange(field: "vColor", lower:
"red", upper: "blue"))
client.search(query: query, completion: {(response, httpResponse, err)-> Void in
        //Handle response
})
```

3. Multilevel

The multilevel filter is used to filter products based on categories. It can be defined in the API call in two ways:

- Using Field Ids

  "CategoryIdFilter" is used to filter the results using category path comprised of category IDs

  Example:

```
let categoryIdFilter = CategoryIdFilter()
categoryIdFilter.categories.append("FA")
categoryIdFilter.categories.append("A0485")
let query = SearchQuery(key: "Shirt", categoryFilter: categoryIdFilter)
client.search(query: query, completion: {(response, httpResponse, err)-> Void in
        //Handle response
})
```

- Using Field Names

  "CategoryNameFilter" is used to filter the results using category path comprised of category Names

  Example:

```
let categoryNameFilter = CategoryNameFilter()
categoryNameFilter.categories.append("Fashion")
categoryNameFilter.categories.append("Shirts")
let query = SearchQuery(key: "Shirt", categoryFilter: categoryNameFilter)
client.search(query: query, completion: {(response, httpResponse, err)-> Void in
        //Handle response
```

```
})
```

L. MULTIPLE FILTERS

There are two types of filter operations:

- **AND**

    1. Using Field IDs

    MultipleIdFilter is takes 2 parameter, field Id and value id.

    Multiple filters can be added and ‘operatorType' is set to ‘AND’.

    Sample:

```
let multipleIdFilter = MultipleIdFilter()
multipleIdFilter.operatorType = .AND
multipleIdFilter.filters.append(IdFilter(field: "76678", value: "5001"))
multipleIdFilter.filters.append(IdFilter(field: "76678", value: "5021"))
let query = SearchQuery(key: "Shirt", multipleFilter: multipleIdFilter)
client.search(query: query, completion: {(response, httpResponse, err)-> Void in
        //Handle response
})
```

    2. Using Field Names

    MultipleNameFilter is takes 2 parameter, field name and value name.

    Multiple filters can be added and ‘operatorType' is set to ‘AND’.

    Sample:

```
let multipleNameFilter = MultipleNameFilter()
multipleNameFilter.operatorType = .AND
multipleNameFilter.filters.append(IdFilter(field: "vColor_uFilter", value: "Black"))
multipleNameFilter.filters.append(IdFilter(field: "vColor_uFilter", value: "White"))
let query = SearchQuery(key: "Shirt", multipleFilter: multipleIdFilter)
client.search(query: query, completion: {(response, httpResponse, err)-> Void in
        //Handle response
})
```

- **OR**

1. Using Field IDs

MultipleIdFilter is takes 2 parameter, field Id and value id.

Multiple filters can be added and ‘operatorType' is set to ‘OR’.

Sample:

```
let multipleIdFilter = MultipleIdFilter()
multipleIdFilter.operatorType = .OR
multipleIdFilter.filters.append(IdFilter(field: "76678", value: "5001"))
multipleIdFilter.filters.append(IdFilter(field: "76678", value: "5021"))
let query = SearchQuery(key: "Shirt", multipleFilter: multipleIdFilter)
client.search(query: query, completion: {(response, httpResponse, err)-> Void in
        //Handle response
})
```

2. Using Field Names

MultipleNameFilter is takes 2 parameter, field name and value name.

Multiple filters can be added and ‘operatorType' is set to ‘OR’.

Sample:

```
 let multipleNameFilter = MultipleNameFilter()
multipleNameFilter.operatorType = .OR
multipleNameFilter.filters.append(IdFilter(field: "vColor_uFilter", value: "Black"))
multipleNameFilter.filters.append(IdFilter(field: "vColor_uFilter", value: "White"))
let query = SearchQuery(key: "Shirt", multipleFilter: multipleIdFilter)
client.search(query: query, completion: {(response, httpResponse, err)-> Void in
        //Handle response
})
```

M. SORT

The sort parameter is used to rank the products based on specified fields in the specified order.

Sort can be done on single field or multiple fields.

- Single field sort

  fieldName: The field on which the sort is applied.

  sortOrder: The order in which the sort is applied. This value can be "ASC" (for ascending) or "DSC" (for descending)

  Sample:

```
var fieldsWithOrder = Array<FieldSortOrder>()
fieldsWithOrder.append(FieldSortOrder(field: "price", order: .ASC))

let query = SearchQuery(key: "Shirt", fieldsSortOrder: fieldsWithOrder)
client.search(query: query, completion: {(response, httpResponse, err)-> Void in
    //Handle response
})
```

- Multiple field sort

  Here 2 or more FieldSortOrder instances are added.

Sample:

```
var fieldsWithOrder = Array<FieldSortOrder>()
fieldsWithOrder.append(FieldSortOrder(field: "price", order: .ASC))
fieldsWithOrder.append(FieldSortOrder(field: "title", order: .DSC))

let query = SearchQuery(key: "Shirt", fieldsSortOrder: fieldsWithOrder)
client.search(query: query, completion: {(response, httpResponse, err)-> Void in
    //Handle response
})
```

Using Auto Suggest Methods

The **Autosuggest** feature provides query suggestions which helps visitors to search faster.

Query method signature:

```
init(withKey:String, format: ResponseFormat = .JSON, inField: DocTypeInField? = nil, keywordSuggestions: DocTypeKeywordSuggestions? = nil, topQueries: DocTypeTopQueries? = nil, promotedSuggestions: DocTypePromotedSuggestions? = nil, popularProducts: DocTypePopularProducts? = nil, variant: Variant? = nil, filter: FilterAbstract? = nil)
```

Autosuggest method signature:

```
func search(query:SearchQuery, completion: @escaping(_ response: Dictionary<String, Any>?,_ httpResponse: HTTPURLResponse?, _ error:Error?)  -> Void) {
        // Handle response
}
```

Below are few sample of invoking autoSuggest() method with arguments.

A. AUTOSUGGEST QUERY

AutoSuggest can be initialised with ‘Key’ for suggestions are expected.

```
        let autoSuggestQuery = AutoSuggestQuery(withKey: "Shir")
        client.autoSuggest(query: autoSuggestQuery, completion: {(response, httpResponse, err) -> Void in
                //Handle response
         })
```

B. VARIANT

Variants can enabled or disabled in AutoSuggest query response.

Variant Status can be set to true/false and shown below.

```
	let autoSuggestQuery = AutoSuggestQuery(withKey: "Shir", variant:
	Variant(has: true))
	client.autoSuggest(query: autoSuggestQuery, completion: {(response,
httpResponse, err) -> Void in
		//Handle response
	})
```

C. DOCTYPE

Unbxd Autosuggest comprises of different types of suggestions that are known as doctypes. A standard Unbxd Autosuggest is segmented into five doctypes:

1. In Field

The query being typed by your visitor can belong to multiple product categories based on your product feed. The **In-fields** doctype in Autosuggest suggest groups of relevant products along with their associated field values the query may belong to.

In Field doctype with result count can be configured as shown below:

```
	let autoSuggestQuery = AutoSuggestQuery(withKey: "Shir", docType:
	DocTypeInField(resultsCount: 3))
	client.autoSuggest(query: autoSuggestQuery, completion: {(response,
httpResponse, err) -> Void in
		//Handle response
	})
```

If resultsCount is not set, default value 2 will be considers as results count for In Field doctype.

2. Keyword Suggestions

These are intelligent suggestions generated by Unbxd cloud servers whose algorithm identifies the keywords from the query being typed and suggests relevant products based on your product feed accordingly.

Keyword Suggestions doctype with result count can be configured as shown below:

```
    let autoSuggestQuery = AutoSuggestQuery(withKey: "Shir", docType:
    DocTypeKeywordSuggestions(resultsCount: 4))
    client.autoSuggest(query: autoSuggestQuery, completion: {(response,
httpResponse, err) -> Void in
        //Handle response
    })
```

If resultsCount is not set, default value 2 will be considers as results count for Keyword Suggestions doctype.

3. Top Queries

As the name suggests, this autosuggest doctype displays the frequently searched queries in your ecommerce store. These top queries are populated with the help of Unbxd Analytics which keeps a track of your store.

Top Queries doctype with result count can be configured as shown below:

```
    let autoSuggestQuery = AutoSuggestQuery(withKey: "Shir", docType:
    DocTypeTopQueries(resultsCount: 3))
    client.autoSuggest(query: autoSuggestQuery, completion: {(response,
httpResponse, err) -> Void in
        //Handle response
    })
```

If resultsCount is not set, default value 2 will be considers as results count for Top Queries doctype.

4. Promoted Suggestions

Promoted Suggestions are documents that a customer can configure directly from merchandising console. This gives you the flexibility to manually insert keyword suggestions in autosuggest which may not be part of the default relevance results.

Promoted Suggestions doctype with result count can be configured as shown below:

```
let autoSuggestQuery = AutoSuggestQuery(withKey: "Shir", docType:
DocTypePromotedSuggestions(resultsCount: 5))
client.autoSuggest(query: autoSuggestQuery, completion: {(response,
httpResponse, err) -> Void in
        //Handle response
})
```

If resultsCount is not set, default value 2 will be considers as results count for Promoted Suggestions doctype.

5. Popular Products

The Popular Products doctype displays popular product in your e-commerce store with thumbnail images.

Popular Products doctype with fields and result count can be configured as shown below:

```
let autoSuggestQuery = AutoSuggestQuery(withKey: "Shir", docType:
DocTypePopularProducts(resultsCount: 3, fields: ["vColor","price"]))
client.autoSuggest(query: autoSuggestQuery, completion: {(response,
httpResponse, err) -> Void in
        //Handle response
})
```

If resultsCount is not set, default value 3 will be considers as results count for Promoted Suggestions doctype.

D. FILTERS

Filters are used in AutoSuggest method to restrict the products based on criteria passed.

Two types of filters are supported in autoSuggestWithQuery() method

1. Text

The text filter is used to filter products based on fields with string values such as color, gender, brand, etc. It can be defined in the API call in two ways:

- Using Field Ids

  IdFilter can be formed with 2 parameters.

  field: The id of the field on which the text filter is applied.

  value: The id of the value on which the results are filtered.

Example:

```
let idFilter = IdFilter(field: "76678", value: "5001")
let autoSuggestQuery = AutoSuggestQuery(withKey: "Shir", docType:
DocTypeKeywordSuggestions(resultsCount: 4), filter: idFilter)
client.autoSuggest(query: autoSuggestQuery, completion: {(response,
httpResponse, err) -> Void in
    //Handle response
})
```

- Using Field Names

  Again NameFilter can be formed with 2 parameters.

  field: The id of the field on which the text filter is applied.

  value: The id of the value on which the results are filtered.

Example:

```
let nameFilter = NameFilter(field: "vColor_uFilter", value: "Black")
let autoSuggestQuery = AutoSuggestQuery(withKey: "Shir", docType:
DocTypeKeywordSuggestions(resultsCount: 4), filter: nameFilter)
client.autoSuggest(query: autoSuggestQuery, completion: {(response,
httpResponse, err) -> Void in
    //Handle response
})
```

2. Range

The range filter is used to filter products based on fields with dataypes - *date*, *number* or *decimal*. It can be defined in the API in two ways:

- Using Field Id

  Filter Range of type id is built using FilterIdRange class and it can be initialised with below parameters.

  field: The id of the field on which the text filter is applied.

  lower: The id of the lower limit of the range.

  upper: The id of the upper limit of the range.

  Example:

```
let idRange = IdFilterRange(field: "76678", lower: "2034", upper: "8906")
let autoSuggestQuery = AutoSuggestQuery(withKey: "Shir", docType:
DocTypeKeywordSuggestions(resultsCount: 4), filter: idRange)
client.autoSuggest(query: autoSuggestQuery, completion: {(response,
httpResponse, err) -> Void in
        //Handle response
})
```

- Using Field Name

  Filter Range of type name is built using FilterNameRange class and it can be initialised with below parapeters.

  field: The name of the field on which the text filter is applied.

  lower: The name of the lower limit of the range.

  upper: The name of the upper limit of the range.

  Example:

```
let nameRange = NameFilterRange(field: "vColor", lower: "red", upper: "blue")
let autoSuggestQuery = AutoSuggestQuery(withKey: "Shir", docType:
DocTypeKeywordSuggestions(resultsCount: 4), filter: nameRange)
```

```
        client.autoSuggest(query: autoSuggestQuery, completion: {(response, httpResponse, err) -> Void in
                //Handle response
        })
```

Using UserId Method

SDK generates user id internally and using below method App can get UserId and visit type.

```
func userId() -> UserId
```

User Id instance would have,

```
let id: String
let visitType: String
```

Getting Request Id

RequestId should be passed to all the analytics api's.

It should be parsed from response header for key "**Unbxd-Request-Id**" from the request to Unbxd framework.

Eg.

```
extension HTTPURLResponse {
    func unbxdRequestId() -> String? {
        if let allHeaders = self.allHeaderFields as? [String: String] {
            if let id = allHeaders["Unbxd-Request-Id"] {
                return id
            }
        }
        return nil
    }
}
```

Using Analytics Method

Using Analytics methods app can publish event details and user would be able track those events in the portal.

Analytics method signature:

```
func track(analyticsDetails:AnalyticsAbstract, completion: @escaping (_ response: Dictionary<String, Any>?,_ httpResponse: HTTPURLResponse?, _ error:Error?) -> Void) {
    // Handle response or request
}
```

Below are the events and argument each event would be accepting.

1. VISITOR

Whenever a user visits the page, a visitor event is fired,

Sample

```
let visitorAnalytics = VisitorAnalytics(uid: userId.id, visitType: userId.visitType,
 requestId: requestId)
client.track(analyticsDetails: visitorAnalytics, completion: {(response, httpResponse, err) -> Void in
        //Handle response
})
```

Note: Uid and visit type can be get by using userId method.

2. SEARCH

A search event is fired when a user types something in the search box and taps on the search button.

Sample

```
let searchAnalytics = SearchAnalytics(uid: userId.id, visitType: userId.visitType,
requestId: requestId, searchKey: "Shirt")
```

```
        client.track(analyticsDetails: searchAnalytics, completion: {(response, httpResponse, err) -> Void in
                //Handle response
        })
```

3. CATEGORY PAGE

Category Page event is fired when a user shops by department and visits a category page on the site.

Sample

```
        let categoryQuery = CategoryNamePath(withCategories:
         ["home","furniture","entrywayfurniture"])
        let categoryPageAnalytics = CategoryPageAnalytics(uid: userId.id, visitType:
         userId.visitType, requestId: requestId, categoryPages: categoryQuery, pageType:
         PageType.CategoryPath)
        client.track(analyticsDetails: categoryPageAnalytics, completion: {(response, httpResponse, err) -> Void in
                //Handle response
        })
```

4. PRODUCT CLICK

Whenever a user clicks on a particular product in the search, recommendations or category page results, a 'click' action is generated.

Sample

```
        let productClickAnalytics = ProductClickAnalytics(uid: userId.id, visitType:
        userId.visitType, requestId: requestId, pID: "2301609", query: "Socks", pageId:
        "Order", boxType: "RECOMMENDED_FOR_YOU")
        client.track(analyticsDetails: productClickAnalytics, completion: {(response, httpResponse, err) -> Void in
                //Handle response
        })
```

5. PRODUCT ADD TO CART

Whenever a user adds a product to his cart, a 'cart' action is generated.

Sample

```
	let addToCartAnalytics = ProductAddToCartAnalytics(uid: userId.id, visitType:
	userId.visitType, requestId: requestId, productID: "2301609", variantId: "231221",
	quantity: 2)
	client.track(analyticsDetails: addToCartAnalytics, completion: {(response,
httpResponse, err) -> Void in
		//Handle response
	})
```

6. PRODUCT ORDER

An 'order' event will be fired after an order completion, when the user comes back to the product page from the payment gateway. If a user buys multiple products in a single order, then multiple order events should be fired.

Sample

```
	let orderAnalytics = ProductOrderAnalytics(uid: userId.id, visitType:
userId.visitType,
	requestId: requestId, productID: "2301609", price: 20.5, quantity: 3)
	client.track(analyticsDetails: orderAnalytics, completion: {(response:Any?,
	error:Error?) -> Void in
		//Handle response
	})
```

7. PRODUCT DISPLAY PAGE VIEW

Product Display Page View is fired when a user visits a Product Display Page.

Sample

```
	let productDisplayPageAnalytics = ProductDisplayPageViewAnalytics(uid:
userId.id,
	visitType: userId.visitType, requestId: requestId, skuId: "2034")
```

```
client.track(analyticsDetails: productDisplayPageAnalytics, completion:
{(response, httpResponse, err)-> Void in
        //Handle response
})
```

8. CART REMOVAL

Cart Removal event is fired when a product in cart is removed..

Sample

```
let cartRemovalAnalytics = CartRemovalAnalytics(uid: userId.id, visitType:
 userId.visitType, requestId: requestId, skuId: "2034", variantId: "231221", quantity:
 2)
client.track(analyticsDetails: cartRemovalAnalytics, completion: {(response,
httpResponse, err) -> Void in
        //Handle response
})
```

9. AUTOSUGGEST

An autosuggest event is fired when a user types something, then clicks on any of autosuggest results shown.

Sample

```
let autoSuggestAnalytics = AutoSuggestAnalytics(uid: userId.id, visitType:
userId.visitType, requestId: requestId, skuId: "2034", query: "Red Socks", docType:
"IN_FIELD", internalQuery: "red", fieldValue: "Red socks", fieldName: "infield1",
sourceField: "color type", unbxdPrank: 6)
client.track(analyticsDetails: autoSuggestAnalytics, completion: {(response,
httpResponse, err) -> Void in
        //Handle response
})
```

Skuid, query, doctype, internalQuery etc are obtained from Autosuggest response data. unbxdPrank is the position from the list of items received in Autosuggest response.

10. RECOMMENDATION WIDGET IMPRESSION

If App is subscribed to Unbxd Recommendations, every time the user clicks on any Recommendation widget, the API below needs to be called.

Sample

```
let recommendationImpressionAnalytics = RecommendationWidgetAnalytics(uid:
userId.id, visitType: userId.visitType, requestId: requestId, recommendationType:
RecommendationType.ViewerAlsoViewed, productIds: ["1692741-
1758","01692015-285","1692908-480"])
client.track(analyticsDetails: recommendationImpressionAnalytics, completion:
{(response, httpResponse, err)-> Void in
    //Handle response
})
```

11. SEARCH IMPRESSION

A search impression event is fired when a search results page loads for the first time, and whenever results changes on applying pagination, autoscroll, sort, and filters. For each of these action, unique Ids of the products visible on search page should be sent as payload.

Sample

```
let searchImpressionAnalytics = SearchImpressionAnalytics(uid: userId.id,
visitType:
userId.visitType, requestId: requestId, query: "Shoes", productIds: ["1692741-
1758","01692015-285","1692908-480"])
client.track(analyticsDetails: searchImpressionAnalytics, completion:
{(response, httpResponse, err)-> Void in
    //Handle response
})
```

12. CATEGORY PAGE IMPRESSION

A Category Page Impression event is fired when a category page results loads for first time, and when ever the results changes on applying pagination, autoscroll, sort, and

filters. For each of these action, unique Ids of the products visible on search page should be sent as payload.

Sample

```
let categoryPathQuery = CategoryNamePath.init(withCategories: ["men","shirt"])
let categoryPageImpression = CategoryPageImpressionAnalytics(uid: userId.id,
visitType: userId.visitType, requestId: requestId, categoryPages:
categoryPathQuery, pageType: PageType.Url, productIds: ["1692741-
1758","01692015-285","1692908-
480"])
client.track(analyticsDetails: categoryPageImpression, completion:
{(response, httpResponse, err)-> Void in
    //Handle response
})
```

13. DWELLTIME

A dwellTime event is used to capture the amount of time spent on product description page.

Sample

```
let dwellTimeAnalytics = DwellTimeAnalytics(uid: userId.id, visitType:
userId.visitType, requestId: requestId, productID: "2301609", dwellTime: 60)
client.track(analyticsDetails: dwellTimeAnalytics, completion: {(response,
httpResponse, err) -> Void in
    //Handle response
})
```

14. FACETS

A facet event is fired when a filter is applied on Search or Category pages.

Sample

```
let facetAnalytics = FacetAnalytics(uid: userId.id, visitType: userId.visitType,
requestId: requestId, searchQuery: "Shirts", facetFields: NameFilter(field: "fit_fq",
value: "Fitted"))
```

```
client.track(analyticsDetails: facetAnalytics, completion: {(response:Any?,
error:Error?) -> Void in
        //Handle response
})
```

Using Browse Methods (Category pages)

Browse methods operates on Category fields query which is configured part of Browse Query.

Query method signature:

init(categoryQuery: CategoryAbstract, rows: Int? = nil, start: Int? = nil, format: ResponseFormat = .JSON, spellCheck: Bool = false, analytics: Bool = false, statsField: String? = nil, variant: Variant? = nil, fields:Array<String>? = nil, facet: Facet = .None, filter: FilterAbstract? = nil, categoryFilter: CategoryFilterAbstract? = nil, multipleFilter: MultipleFilterAbstract? = nil, fieldsSortOrder: Array<FieldSortOrder>? = nil, personalization: Bool? = nil)

Browse method signature:

func browse(query:BrowseQuery, completion: @escaping (_ response: Dictionary<String, Any>?,_ httpResponse: HTTPURLResponse?, _ error:Error?) -> Void)

Lets see how Browse Query can be composed and passed in browse() method invocation.

A. BROWSEQUERY

BrowseQuery consists of Category path or field details parameter

Query with Category can be build as shown below:

- Using Field Ids

```
let categoryQuery = CategoryIdPath(withCategories: ["FA","FA0484"])
let browseQuery = BrowseQuery(categoryQuery: categoryQuery)
client.browse(query: browseQuery, completion: {(response, httpResponse, err)->
    Void in
    //Handle response
```

```
})
```

- Using Field Names

```
let categoryQuery = CategoryNameFields(field: "vColor", value:      “Black")
let browseQuery = BrowseQuery(categoryQuery: categoryQuery)
client.browse(query: browseQuery, completion: {(response, httpResponse, err)->
        Void in
        //Handle response
})
```

- Using Page Ids

```
let categoryQuery = CategoryIdPath(withCategories: [“FA”,"FA0484"])
let browseQuery = BrowseQuery(categoryQuery: categoryQuery)
client.browse(query: browseQuery, completion: {(response, httpResponse, err)->
        Void in
        //Handle response
})
```

- Using Page Names

```
let categoryQuery = CategoryNamePath(withCategories: ["cat120009"])
let browseQuery = BrowseQuery(categoryQuery: categoryQuery)
client.browse(query: browseQuery, completion: {(response, httpResponse, err)->
        Void in
        //Handle response
})
```

B. FORMAT

The format parameter specifies the format of the response. Possible values are ‘json’ or ‘xml’. It is an optional parameter and the default value is ‘json’.

```
let browseQuery = BrowseQuery(categoryQuery: categoryQuery)
, format: .XML)
```

```
client.browse(query: browseQuery, completion:{(response, httpResponse, err)->
Void in
        //Handle response
})
```

C. START

The start parameter is used to offset the results by a specific number. It indicates offset in the complete result set of the products. It is an optional parameter and the default value is 0.

```
let browseQuery = BrowseQuery(categoryQuery: categoryQuery)
, start: 2, format: .JSON)
client.browse(query: browseQuery, completion: {(response, httpResponse, err)->
        Void in
         //Handle response
})
```

D. ROWS

The rows parameter is used to paginate the results of a query. It indicates number of products in a single page. It is an optional parameter and the default value is 10, maximum value is 100.

```
let browseQuery = BrowseQuery(categoryQuery: categoryQuery)
, rows: 20)
client.browse(query: browseQuery, completion: {(response, httpResponse, err)->
        Void in
         //Handle response
})
```

E. SPELLCHECK

The spellcheck feature provides spelling suggestions or spell-check for misspelled search queries.

```
let browseQuery = BrowseQuery(categoryQuery: categoryQuery)
, spellCheck: true)
```

```
client.browse(query: browseQuery, completion: {(response, httpResponse, err)->
        Void in
        //Handle response
})
```

F. ANALYTICS

The analytics parameter enables or disables tracking the query hit for analytics. By default, tracking is enabled.

```
let browseQuery = BrowseQuery(categoryQuery: categoryQuery)
, analytics: false)
client.browse(query: browseQuery, completion: {(response, httpResponse, err)->
        Void in
         //Handle response
 })
```

G. STATS

The stats parameter gives information about the products with highest and lowest field value.

```
let browseQuery = BrowseQuery(categoryQuery: categoryQuery)
, statsField: "price")
client.browse(query: browseQuery, completion: {(response, httpResponse, err)->
        Void in
         //Handle response
 })
```

H. VARIANTS

Variants parameter enables or disables variants in the API response. It can take two values: “true” or “false”. Default value is “false”.

```
let browseQuery = BrowseQuery(categoryQuery: categoryQuery)
, variant: Variant(has: true, count: 2))
```

```
client.browse(query: browseQuery, completion: {(response, httpResponse, err)->
        Void in
         //Handle response
 })
```

If you want to get more than one variants in the API response, you can use ‘variantCount’ parameter. It can have any numerical value (eg, 1,2,3,etc) or “.max” (to get all the variants).

I. FIELDS

The fields parameter is used to specify the set of fields to be returned. When returning the results, only fields in the list will be included.

```
let browseQuery = BrowseQuery(categoryQuery: categoryQuery)
, fields: ["title","vPrice"])
client.browse(query: browseQuery,, completion: {(response, httpResponse, err)->
Void in
        //Handle response
})
```

J. FACETS

Facets are the filters in the UI that allow visitors to narrow down result set based on product fields. Facet option can have 3 values:

1.**Multi-level**: The facer multilevel parameter is used to enable multi-level facets in the API response.

```
let browseQuery = BrowseQuery(categoryQuery: categoryQuery)
, facet: .MultiLevel)
client.browse(query: browseQuery,, completion: {(response, httpResponse, err)->
Void in
    //Handle response
 })
```

2.**Multi-select**: This feature enables or disables the option to select multiple values within a facet or across facets for visitors.

```
let browseQuery = BrowseQuery(categoryQuery: categoryQuery)
, facet: .MultiSelect)
client.browse(query: browseQuery, completion: {(response, httpResponse, err)->
        Void in
         //Handle response
})
```

3.**Selected**: The selected facet parameter enables or disables the Selected Facets in the API response.

Selected facet with field id and value id:

```
let browseQuery = BrowseQuery(categoryQuery: categoryQuery),
facet: .Selected(IdFilter(field: "76678", value: "5001")))
client.browse(query: browseQuery, completion: {(response, httpResponse, err)->
        Void in
         //Handle response
})
```

Selected facet with field name and value name:

```
let browseQuery = BrowseQuery(categoryQuery: categoryQuery),
facet: .Selected(NameFilter(field: "Brand_uFilter", value: "Vince Camuto")))
client.browse(query: browseQuery, completion: {(response, httpResponse, err)->
        Void in
         //Handle response
})
```

K. FILTERING

Filtering can be performed on fields using field Id or field Name.

Three types of filters are supported:

1. **Text**

The text filter is used to filter products based on fields with string values such as color, gender, brand, etc. It can be defined in the API call in two ways:

- Using Field Ids

  IdFilter can be formed with 2 parameters.

  field: The id of the field on which the text filter is applied.

  value: The id of the value on which the results are filtered.

Example:

```
let browseQuery = BrowseQuery(categoryQuery: categoryQuery), filter:
IdFilter(field: "76678", value: "5001"))
client.browse(query: browseQuery, completion: {(response, httpResponse, err)->
        Void in
         //Handle response
})
```

- Using Field Names

  Again NameFilter can be formed with 2 parameters.

  type: The id of the field on which the text filter is applied.

  value: The id of the value on which the results are filtered.

Example:

```
let browseQuery = BrowseQuery(categoryQuery: categoryQuery), filter:
NameFilter(field: "vColor_uFilter", value: "Black"))
 client.browse(query: browseQuery, completion: {(response, httpResponse, err)->
Void in
        //Handle response
 })
```

2. Range

The range filter is used to filter products based on fields with dataypes - *date*, *number* or *decimal*. It can be defined in the API in two ways:

- Using Field Id

  Filter Range of type id is built using IdFilterRange class and it can be initialised with below parameters.

  field: The id of the field on which the text filter is applied.
  lower: The id of the lower limit of the range.
  upper: The id of the upper limit of the range.

  Example:

```
let browseQuery = BrowseQuery(categoryQuery: categoryQuery), filter:
IdFilterRange(field: "76678", lower: "2034", upper: "8906"))
client.browse(query: browseQuery, completion: {(response, httpResponse, err)->
        Void in
         //Handle response
})
```

- Using Field Name

  Filter Range of type name is built using NameFilterRange class and it can be initialised with below parameters.

  field: The name of the field on which the text filter is applied.
  lower: The name of the lower limit of the range.
  upper: The name of the upper limit of the range.

  Example:

```
let query = SearchQuery(key: "Shirt", filter: NameFilterRange(field: "vColor", lower:
"red", upper: "blue"))
client.browse(query: browseQuery, completion: {(response, httpResponse, err)->
        Void in
         //Handle response
})
```

3. Multilevel

The multilevel filter is used to filter products based on categories. It can be defined in the API call in two ways:

- Using Field Ids

  “CategoryIdFilter” is used to filter the results using category path comprised of category IDs

  Example:

```
let categoryIdFilter = CategoryIdFilter()
categoryIdFilter.categories.append("FA")
categoryIdFilter.categories.append("A0485")
let browseQuery = BrowseQuery(categoryQuery: categoryQuery),
categoryFilter: categoryIdFilter)
client.browse(query: browseQuery, completion: {(response, httpResponse, err)->
        Void in
         //Handle response
})
```

- Using Field Names

  “CategoryNameFilter” is used to filter the results using category path comprised of category Names

  Example:

```
let categoryNameFilter = CategoryNameFilter()
categoryNameFilter.categories.append("Fashion")
categoryNameFilter.categories.append("Shirts")
let browseQuery = BrowseQuery(categoryQuery: categoryQuery),
categoryFilter: categoryNameFilter)
client.browse(query: browseQuery, completion: {(response, httpResponse, err)->
        Void in
         //Handle response
})
```

L. MULTIPLE FILTERS

There are two types of filter operations:

- **AND**

    1. Using Field IDs

    MultipleIdFilter is takes 2 parameter, field Id and value id.

    Multiple filters can be added and ‘operatorType' is set to ‘AND’.

    Sample:

```
let multipleIdFilter = MultipleIdFilter()
multipleIdFilter.operatorType = .AND
multipleIdFilter.filters.append(IdFilter(field: "76678", value: "5001"))
multipleIdFilter.filters.append(IdFilter(field: "76678", value: "5021"))
let browseQuery = BrowseQuery(categoryQuery: categoryQuery), multipleFilter:
multipleIdFilter)
client.browse(query: browseQuery, completion: {(response, httpResponse, err)->
        Void in
         //Handle response
})
```

    2. Using Field Names

    MultipleNameFilter is takes 2 parameter, field name and value name.

    Multiple filters can be added and ‘operatorType' is set to ‘AND’.

    Sample:

```
let multipleNameFilter = MultipleNameFilter()
multipleNameFilter.operatorType = .AND
multipleNameFilter.filters.append(IdFilter(field: "vColor_uFilter", value: "Black"))
multipleNameFilter.filters.append(IdFilter(field: "vColor_uFilter", value: "White"))
let browseQuery = BrowseQuery(categoryQuery: categoryQuery), multipleFilter:
multipleIdFilter)
client.browse(query: browseQuery, completion: {(response, httpResponse, err)->
        Void in
         //Handle response
})
```

- **OR**

    1. Using Field IDs

MultipleIdFilter is takes 2 parameter, field Id and value id.
Multiple filters can be added and ‘operatorType' is set to ‘OR’.

Sample:

```
let multipleIdFilter = MultipleIdFilter()
multipleIdFilter.operatorType = .OR
multipleIdFilter.filters.append(IdFilter(field: "76678", value: "5001"))
multipleIdFilter.filters.append(IdFilter(field: "76678", value: "5021"))
let browseQuery = BrowseQuery(categoryQuery: categoryQuery), multipleFilter:
multipleIdFilter)
client.browse(query: browseQuery, completion: {(response, httpResponse, err)->
        Void in
         //Handle response
})
```

2. Using Field Names
MultipleNameFilter is takes 2 parameter, field name and value name.
Multiple filters can be added and ‘operatorType' is set to ‘OR’.

Sample:

```
 let multipleNameFilter = MultipleNameFilter()
multipleNameFilter.operatorType = .OR
multipleNameFilter.filters.append(IdFilter(field: "vColor_uFilter", value: "Black"))
multipleNameFilter.filters.append(IdFilter(field: "vColor_uFilter", value: "White"))
let browseQuery = BrowseQuery(categoryQuery: categoryQuery), multipleFilter:
multipleIdFilter)
client.browse(query: browseQuery, completion: {(response, httpResponse, err)->
        Void in
         //Handle response
})
```

M. SORT

The sort parameter is used to rank the products based on specified fields in the specified order.
Sort can be done on single field or multiple fields.

- Single field sort

  fieldName: The field on which the sort is applied.

  sortOrder: The order in which the sort is applied. This value can be "ASC" (for ascending) or "DSC" (for descending)

  Sample:

```
var fieldsWithOrder = Array<FieldSortOrder>()
fieldsWithOrder.append(FieldSortOrder(field: "price", order: .ASC))

let browseQuery = BrowseQuery(categoryQuery: categoryQuery),
fieldsSortOrder: fieldsWithOrder)
client.browse(query: browseQuery, completion: {(response, httpResponse, err)->
    Void in
    //Handle response
})
```

- Multiple field sort

  Here 2 or more FieldSortOrder instances are added.

Sample:

```
var fieldsWithOrder = Array<FieldSortOrder>()
fieldsWithOrder.append(FieldSortOrder(field: "price", order: .ASC))
fieldsWithOrder.append(FieldSortOrder(field: "title", order: .DSC))

let browseQuery = BrowseQuery(categoryQuery: categoryQuery),
fieldsSortOrder: fieldsWithOrder)
client.browse(query: browseQuery, completion: {(response, httpResponse, err)->
    Void in
    //Handle response
})
```

Using Recommendations Widget

Unbxd Recommendations offers a wide-range of widgets for every page. Recommendations API returns product details for options like MoreLikeThis, RecentlyViewed etc.

Recommendations method signature:

```
func recommend(recommendationQuery:RecommendationQuery, completion: @escaping (_ response: Dictionary<String, Any>?,_ httpResponse: HTTPURLResponse?, _ error:Error?) -> Void) {
    // Handle response or request
}
```

Below are the different recommendation types and its arguments.

1. RECOMMENDED FOR YOU

The Recommended For You method returns recommendations based on the visitor's interaction history on the online store or app.

Sample:

```
let uid = "1409753998404-35253"

let forYouQuery = RecommendedForYourRecomendations(uid: uid, region: "USA", currency: "USD", format: .JSON)
client.recommend(recommendationQuery: forYouQuery, completion:
{(response, httpResponse, err)-> Void in
    //Handle response
})
```

Note: UID should be unique string value which is generated by app and retained through out the application life.

- region : ISO standard country code.
- currencyFormat: ISO standard currency code.
- responseFormat: .JSON or XML

2. RECENTLY VIEWED

The Recently Viewed method recommends products that were recently viewed by a visitor.

Sample:

```
let recentlyViewedQuery = RecentlyViewedRecomendations(uid: uid, productID: "2312314", region: "USA")
client.recommend(recommendationQuery: recentlyViewedQuery, completion: {(response, httpResponse, err)-> Void in
    //Handle response
})
```

3. MORE LIKE THESE

The "More Like These" method is built to recommend products similar to the one being viewed on the PDP (Product Detail Page).

Sample:

```
let moreLikeThisQuery = MoreLikeThisRecomendations(uid: uid, productID: "2312314", region: "USA")
client.recommend(recommendationQuery: moreLikeThisQuery, completion: {(response, httpResponse, err)-> Void in
    //Handle response
})
```

4. VIEWED ALSO VIEWED

As the name suggests, this method recommends products viewed by other visitors.
Sample:

```
let alsoViewedQuery = ViewedAlsoViewedRecomendations(uid: uid, productID: "2312314", region: "USA")
```

```
client.recommend(recommendationQuery: alsoViewedQuery, completion:
{(response, httpResponse, err)-> Void in
        //Handle response
})
```

5. BOUGHT ALSO BOUGHT

Similar to the "Viewed also Viewed" method, the Bought also Bought method recommends products bought by other visitors.

Sample:

```
let alsoBoughtdQuery = BoughtAlsoBoughtRecomendations(uid: uid,
productID: "2312314", region: "USA")
client.recommend(recommendationQuery: alsoBoughtdQuery, completion:
{(response, httpResponse, err)-> Void in
        //Handle response
})
```

6. CART RECOMMENDATIONS

This method recommends complementary products on the "Cart page" for those present in the visitor's cart.

Sample:

```
Version 1:
let cartRecommendationQuery = CartRecomendations(uid: uid, region: “USA”)
client.recommend(recommendationQuery: cartRecommendationQuery,
completion:   {(response, httpResponse, err)-> Void in
        //Handle response
})

Version 2:
let cartRecommendationQuery = CartRecomendations(uid: uid, region: “USA”,
widget: .Widget1,  version: .Version2)
```

```
client.recommend(recommendationQuery: cartRecommendationQuery,
completion:  {(response, httpResponse, err)-> Void in
        //Handle response
})
```

7. HOMEPAGE TOP SELLERS

This method recommends top selling products bought from the homepage.

Sample:

Version 1:

```
let homePageTopSellerQuery = HomePageTopSellersRecomendations(uid: uid,
region: "USA")
client.recommend(recommendationQuery: homePageTopSellerQuery, completion:
{(response, httpResponse, err)-> Void in
        //Handle response
})
```

Version 2:

```
let homePageTopSellerQuery = HomePageTopSellersRecomendations(uid: kUid,
widget: .Widget2, version: .Version2)
client.recommend(recommendationQuery: homePageTopSellerQuery, completion:
{(response, httpResponse, err)-> Void in
        //Handle response
})
```

8. CATEGORY TOP SELLERS

This method recommends top selling products from a specific category.

Sample:

Version 1:

```
let categoryTopSellerQuery = CategoryTopSellersRecomendations(uid: uid,
region: "USA", categoryName: "<CategoryName>")
client.recommend(recommendationQuery: categoryTopSellerQuery, completion:
{(response, httpResponse, err)-> Void in
        //Handle response
})
```

Version 2:

```
let categoryTopSellerQuery = CategoryTopSellersRecomendations(uid: uid,
categoryLevelNames: ["men" "Tops" "Performance Dress Shirts" "Extra Slim
Shirts",  widget: .Widget2, version: .Version2)
client.recommend(recommendationQuery: categoryTopSellerQuery, completion:
{(response, httpResponse, err)-> Void in
        //Handle response
})
```

9. PDP TOP SELLERS

This method recommends top selling products from a specific product.

Sample:

Version 1:

```
let pdpTopSellerQuery = PDPTopSellersRecomendations(uid: uid, productID:
“<product_id>”)
client.recommend(recommendationQuery: pdpTopSellerQuery, completion:
{(response, httpResponse, err)-> Void in
         //Handle response
})
```

Version 2:

```
let pdpTopSellerQuery = PDPTopSellersRecomendations(uid: uid, productID:
“<product_id>”, version: .Version2)
client.recommend(recommendationQuery: pdpTopSellerQuery, completion:
{(response, httpResponse, err)-> Void in
```

```
        //Handle response
})
```

10. BRAND TOP SELLERS

This method recommends top selling products from a specific brand.

Sample:

```
Version 1:
let brandTopSellerQuery = BrandTopSellersRecomendations(uid: uid, region: "USA", brandName: "Nike")
client.recommend(recommendationQuery: brandTopSellerQuery, completion: {(response, httpResponse, err)-> Void in
        //Handle response
})

Version 2:
let brandTopSellerQuery = BrandTopSellersRecomendations(uid: uid, region: "USA", brandName: “Nike”, version: .Version2)
client.recommend(recommendationQuery: brandTopSellerQuery, completion: {(response, httpResponse, err)-> Void in
        //Handle response
})
```

11. COMPLETE THE LOOK

The Complete the Look method showcases curated products on a Product Display Page (PDP) that are usually associated with the original product.

Sample:

```
let completeTheLookQuery = CompleteTheLookRecomendations(uid: uid, productID: "2312314", region: "USA")
client.recommend(recommendationQuery: completeTheLookQuery, completion: {(response, httpResponse, err)-> Void in
        //Handle response
})
```

VERSION 2 API's

1. Recommendation Widget API

A. Home Page

This method recommends top selling products bought from the homepage.

Sample:

```
let homePageRecommendations = RecommendationsV2(pageType: .Home, uid:
"<uid>", widget: .Widget1)
client.recommend(recommendationQuery: homePageRecommendations,
completion:  {(response, httpResponse, err)-> Void in
        //Handle response
})
```

B. Category Page

This method recommends top selling products from a specific category.

Sample:

```
let categoryPageRecommendations =
RecommendationsV2(pageType: .Cateogory, uid:"<uid>", categoryLevelNames:
        ["men","Tops","Performance Dress Shirts","Extra Slim Shirts"])
client.recommend(recommendationQuery: categoryPageRecommendations,
completion:  {(response, httpResponse, err)-> Void in
        //Handle response
})
```

C. Product Page

This method recommends top selling products from a specific product.

Sample:

```
let productPageRecommendations = RecommendationsV2(pageType: .Pdp, uid:
"<uid>", id: "<product_id>", widget: .Widget1)
client.recommend(recommendationQuery: productPageRecommendations,
        completion:  {(response, httpResponse, err)-> Void in
```

```
        //Handle response
    })
```

D. Brand Page

This method recommends top selling products from a specific brand.

Sample:

```
    let brandPageRecommendations = RecommendationsV2(pageType: .Brand, uid:
    “<uid>”, brand: “<brand-name>")
    client.recommend(recommendationQuery: brandPageRecommendations,
    completion:  {(response, httpResponse, err)-> Void in
        //Handle response
    })
```

E. Cart Page

This method recommends complementary products on the "Cart page" for those present in the visitor's cart.

Sample:

```
    let cartPageRecommendations = RecommendationsV2(pageType: .Cart, uid:
“<uid>”, widget: .Widget1)
    client.recommend(recommendationQuery: cartPageRecommendations,
    completion:  {(response, httpResponse, err)-> Void in
        //Handle response
    })
```

2. Analytics API

A. Product Click Analytics

Whenever a user clicks on a particular product in the search, recommendations or category page results, a 'click' action is generated.

Sample

```
        let productClickAnalytics = ProductClickAnalyticsV2(uid: userId.id, visitType: userId.visitType, requestId: requestId, pID: "2301609", query: "Socks", pageId: "Order", pageType: .Home)
                client?.track(analyticsDetails: productClickAnalytics, completion: { (response, httpResponse, err) in
          /Handle response
      })
```

B. Recommendation Impression Analytics

If App is subscribed to Unbxd Recommendations, every time the user clicks on any Recommendation widget, the API below needs to be called.

Sample

```
        let recommendationImpressionAnalytics =
        RecommendationWidgetAnalyticsV2(uid: userId.id, visitType: userId.visitType,
        requestId: requestId, pageType: .Home, widget: .Widget1, productIds:
        ["1692741-1758","01692015-285","1692908-480"])
        client.track(analyticsDetails: recommendationImpressionAnalytics, completion:
        {(response, httpResponse, err)-> Void in
                //Handle response
        })
```