

## Getting Started With SMS API

(ver. from 8/31/2025)

### Table of Contents.

1. API Library stuff.	1
2. Client Context.	2
3. Data Contract classes.	2
4. Service Contract classes.	2
5. Data exchange with US Customs.	4
5.1 Basic framework.	4
5.2 Documents and directories.	5
5.3 ABI queries.	6
6. Data validation.	7
6.1 Document and directory exceptions handling.	7
6.2 ABI query exceptions handling.	8
6.3 Possible Values.	8

*The purpose of this document is to give only a general overview of API.*

### 1. API Library stuff.

All API classes and interfaces are contained in a single SMS.Broker.Standard.dll file. It was developed under .NET Framework 4.8.

Shortly, SMS.Broker.Standard.dll file includes:

1. DataContext class - It forms an environment with the underlying database. Any working session begins from its initialization.
2. Document classes (descendants of [EntityDocument](#) class) – classes those represent documents ([CustomsEntry](#), [Shipment](#), [ImporterSecurityFiling](#), [Drawback](#), [In-Bond](#) and so on).
3. Directory classes (descendants of [EntityDirectory](#) class) – classes those represent changeable but non-document information ([Contact](#), [ContactBondInfo](#), [CustomsPort](#), [HarmonizedTariff](#) and many others).
4. List items – seldom changed, “conventionally fixed” info. ([Country](#), [Currency](#), [LoadType](#), [ChargeType](#) and many others). Items are combined to respective lists and can be used as Possible Values.
5. Manager interfaces – they represent business logic classes on a client side and allow to get an access to objects of related document or directory class – to create, modify, validate, save them and run some other methods (in particular to send messages to US Customs).

For example, [IContactManager](#) interface allows to create a new object of [Contact](#) class, save it to database, modify it, send contact-related forms to Customs. The same, [ICustomsEntryManager](#) works with objects of [CustomsEntry](#) class.



6. Fault classes – they are necessary to process the validation results of incorrect data.
7. Other classes those we mention in this document and will detail in other documentation.

## 2. Client Context.

Starting the coding, first of all we need to initialize a client context. It can be done by `InitializeContext()` method:

`SMS.Broker.ClientContext.InitializeContext(url, username + "@" + database, password)`, where  
`string` url – service url (for example, `@https://smstest.logisticaldatasolutions.com:8130/brokerservice`),  
`string` username – user name,  
`string` database – database name,  
`string` password – password.

## 3. Data Contract classes.

These classes take a part in a data transmission between client and server parts of SMS System. In this document we will consider three types of them: document, directory and fault classes.

In general case each object of document or directory class corresponds to one header record and a set of related child records on MS SQL Server database. For example, you created an object of `CustomsEntry` class and saved it on server side. It means that some records were added to database: one header record to `CustomsEntries` table, any related records to `Shipments` table, any records to `ShipmentInvoices` table, to `ShipmentsArticle` and `ShipmentsArticleTariffs` tables. After save operation corresponded records appear in database with unique Id numbers.

To create a new object we can use a `new` operator:

```
CustomsEntry myEntry = new CustomsEntry();  
Contact myContact = new Contact();
```

but it is more rational is to use a manager interface that we describe on the next (4. Service Contracts) part. Each document or directory object has a lot of properties and collections those must be filled before validation. Manager fills a part of them by default values while creating (for example, `EntryType = "01"`, `BondType = "9"`, Current Date, Number, default Importer, default `EntryFilerCode`).

Fault class will be described later in (5. Data validation) part.

## 4. Service Contract classes.

These are manager classes (represented on client side by interfaces). They are designed to work with data contract classes: to create, save, read their objects from database and make with them some another operations.

We can create an object of manager class using the manager interface if we already initialized client context (See 2. *Client Context*).

```
ICustomsEntryManager mngrCE =  
ClientContext.ServicesFactory.GetManager<SMS.Broker.ServiceContracts.Documents.ICustomsEntryManager>();
```

```
IContactManager mngrContact =  
ClientContext.ServicesFactory.GetManager<SMS.Broker.ServiceContracts.Directories.IContactManager>();
```

```
IShipmentManager mngrShipment =  
ClientContext.ServicesFactory.GetManager<SMS.Broker.ServiceContracts.Documents.IShipmentManager>();
```



After a manager object was created, manager interface methods are available.

Now we can create a new object of some data contract class:

```
CustomsEntry newCE = mngrCE.New(null);    // Instead of null we can use long Id of another customs entry as a source.
or
Contact Importer = mngrContact.New(null);
```

*Note 1. Navigation properties.*

*If you are familiar with Microsoft Entity Framework navigation property notion you can omit the follow reference, if not, please look at Annex. Getting Started With SMS API, 2. Navigation Properties.*

Initially these objects have only default properties filled. You should add a necessary info by means of user interface or another automated process. For example, it can look like that (it is suggested that variables with your data are on the right side of “=” sign):

```
newCE.EstimatedEntryDate = EstimatedEntryDate;
newCE.DateOfImport = DateOfImport;
newCE.DateOfArrival = DateOfArrival;
newCE.BrokerReferenceNumber = newCE.EntryNumber;
...
newCE.PortOfUnlading = PortOfUnlading;
newCE.ConveyanceName = ConveyanceName;
newCE.TripIdentifier = TripIdentifier;
newCE.PaymentType = PaymentType;
newCE.PreliminaryStatementPrintDate = PreliminaryStatementPrintDate;
...
Shipment shipment = mngrShipment.New(null);
shipment.Carrier_Id = CarrierId;
shipment.Importer_Id = ImporterId;
...
// Add a shipment to new CE:
newCE.Shipments.Add(shipment);
```

We don't provide all customs entry class properties and collections here. There is a lot of. You can see detailed sample in *02. Annex. Getting Started With SMS API.pdf, 2. How to create and fill a new customs entry document.*

Suppose we have added all necessary data. Here we can save our object to database:

```
newCE = mngrCE.Save(newCE);
```

Before the save operation a newCE object has an Id equal to zero and Save() method returns the same object but already with unique Id assigned by SQL Server.

And finally we can read objects from database. There are number of methods to do it.

We can get an object by its Id:

```
CustomsEntry CE = mngrCE.Get(entryId, "");
```

If we want to get also navigation properties and collections included to object (for example Importer and Shipments) we need to add a **string** include parameter equal to **"Importer,Shipments"**:

```
string include = "Importer,Shipments";
CustomsEntry CE = mngrCE.Get(entryId, include);
```

Besides a unique Id an each directory class object saved in database has a **string** Code property. There is a method that uses it for search:

```
string ImporterCode = "ADMD02";
Importer = mngrContact.GetByCode(ImporterCode, "");
```



For more complex requests you can apply [EntityPageQuery](#) class:

```
EntityPageQuery q = new EntityPageQuery();
q.Include = "Importer";           // Read from database also property Importer by Importer_Id (Entity Framework "include")
q.PageSize = 10;                  // Get first 10 records (the big value can decrease an operation performance or cause a timeout exception)
q.QueryTotalCount = true;         // The result will have a total number of records (It requires an additional SQL query and decrease the
                                  // operation performance)
q.RequestFullList = false;        // Get all records (It is only for small tables)
q.Position = 0;                   // Position of the first record in query result.//Position of first record in result of query
q.Navigation = NavigationType.Refresh; // Request will return a list starting with q.Position
q.Order.Add(new OrderItem() { Name = "Number", IsDesc = true }); // Order by Number descending

// Filter Id < 20
q.Criteria = "[Id] < 20L";

// Make a request:
List<CustomsEntry> entries = mgrCE.GetPage(q).Entities;

// Use a list of entries for Web GridView:
System.Web.UI.WebControls.GridView GridView1;
GridView1.DataSource = entries;
```

Detailed description of [EntityPageQuery](#) class you can find in *05. Common usage classes and interfaces. Part 1.pdf*.

More detailed description of customs entry document you can see in *03. Customs entry classes and interfaces. Part 1.pdf* document.

## 5. Data Exchange with US Customs.

### 5.1 Basic framework.

Now it is a time to tell about data exchange with Customs. Simply saying the main idea is that client puts a message to queue on SMS-server side (a new [ABITransmission](#) class record appears on the server) and then SMS-server resends the message to Customs already in "Customs native" form. Messages incoming from Customs also become records on SMS-server side and then can be obtained by client. Besides that incoming message adds new (current) status as last status and new event to related objects.

*Note 2. LastStatuses.*

Detailed info about [LastStatus](#) class methods and properties please see in *05. Common usage classes and interfaces. Part 1.pdf*, 2. Entity Last Status. During its lifecycle document or directory object can undergo different changes of its state which are the result of actions performed on the document. For example, [CustomsEntry](#) class object can be "calculated" (it means that duty and fee is calculated), put to queue, sent as some Customs application to Customs, accepted/rejected by Customs, then Customs sends a response, the document state again is changed and so on. [LastStatus](#) class object is used to have an actual current state of a document. Each document has [LastStatuses](#) property - it is a collection of [LastStatus](#) objects because one [LastStatus](#) object is not enough:

on first, last statuses can differ by their type (i.e. by their [Status](#) property) and the result of actions of a different type must not be lost after the next action result has appeared. For example, the [CustomsEntry](#) object doesn't stop to be "calculated" after it was put to queue ([Status](#) = [EntityStatusType.DocumentCalculated](#) and [Value](#) = [EntityStatusValue.Yes](#) and the next [Status](#) = [EntityStatusType.ABIAppeal](#) and [Value](#) = [EntityStatusValue.ABIAppealToQueue](#)). On this case new Last Status object with [Status](#) = [EntityStatusType.ABIAppeal](#) is added to [LastStatuses](#) collection).

on second, one and the same document can be used to send to Customs the different Customs applications. For example, [CustomsEntry](#) document is used to send Cargo Release (SE) and Entry Summary (AE) applications. And related action results for different applications in a document's [LastStatuses](#) collection must not be overlapped. To differ last statuses of different applications for the same document the [Subject](#) property is applied. On our example [Subject](#) is equal to "SE" and "AE" respectively.

Other words, a combination of [Status](#) and [Subject](#) properties must be unique in [LastStatuses](#) collection of a given object.

*Sample.*

Step 1. Duty and fee were calculated for some [CustomsEntry](#) document. New [LastStatus](#) object with [Status](#) = [EntityStatusType.DocumentCalculated](#) and [Value](#) = [EntityStatusValue.Yes](#) is added to [LastStatuses](#) collection of the document.

Step 2. Then Entry Summary (AE) application was put to queue to be sent to Customs. New [LastStatus](#) object with [Status](#) = [EntityStatusType.ABIAppeal](#), [Value](#) = [EntityStatusValue.ABIAppealToQueue](#) is added to [LastStatuses](#) collection of the document.

Step 3. SMS server processed queue and sent Entry Summary (AE) application to Customs. New last status [Value](#) [EntityStatusValue.ABIAppealSent](#) has the same type [EntityStatusType.ABIAppeal](#) as already added [LastStatus](#) object, [Subject](#) "AE" is the same also. New [LastStatus](#) object is not added to [LastStatuses](#) collection of the document. [Value](#) of the same type object is changed from [EntityStatusValue.ABIAppealToQueue](#) to new [EntityStatusValue.ABIAppealSent](#).



Step 4. Customs sent a response. Document was accepted and hence has a status on Customs side. It is one more type of [LastStatus](#) object. As a result a new [LastStatus](#) object with `Status = EntityStatusType.ABIAAppCustoms` and `Value = EntityStatusValue.ABIAAppCustomsAccepted` is **added** to `LastStatuses` collection of the document.

Note 3. Entity link.

It is an important notion that will come in useful. Please look at 02. Annex. Getting Started With SMS API.pdf, 4. Entity link.

Note 4. Events.

Document or directory class objects that have `LastStatuses` collection have also related events ([EntityEvent](#) objects). They represent a history log of related object but unlike `LastStatuses` it is not included to related object as collection property and available by object's entity link. Event object contains a free form text and in this meaning it is more user-oriented than `LastStatus` object. More info about [EntityEvent](#) class you can see in 05. Common usage classes and interfaces. Part 1.pdf, 2. Entity Event. How to obtain a list of events related to object you can see in 02. Annex. Getting Started With SMS API.pdf, 5. How to get a list of events. (As a sample)

## 5.2 Documents and directories.

Suppose, we have [Contact](#) object already saved on SMS-server and want to send CBPF 5106 (Importer/Consignee create/update) to Customs. We create a manager:

```
IClientManager mngrContact =  
ClientContext.ServicesFactory.GetManager<SMS.Broker.ServiceContracts.Directories.IContactManager>();
```

and then put the message to queue using object's Id:

```
mngrContact.PutToQUEUE(ContactId, Contact.ActionQUEUE.ImporterConsigneeCreate);
```

There are two types of put to queue methods: synchronous `PutToQUEUE()` and asynchronous `PutToQUEUEOneWay()`. Some documents require a time for their processing. To avoid a "frozen" user interface on client side while validation process is running some put to queue methods was built as asynchronous. For more detailed info, please look at 02. Annex. Getting Started With SMS API.pdf, 3. Put to queue methods and their parameters.

The most general way to get Customs response messages from SMS-server is to use an object's link. Every [ABITransmission](#) record that appears on SMS-server side has a link to its source object record. Source object here is a document (or directory) class object for which the message was sent/received. For example, we send a CBPF 7501 for [CustomsEntry](#) object with Id=78. After we put a message to queue the outgoing ABI transmission record is added to SMS-Server and the message is sent to Customs. Then Customs sends a response message and it becomes a new incoming ABI transmission record on SMS-Server. [CustomsEntry](#) object is a source object here. Both ABI transmission records have a link to it: `Entity.Link = "CustomsEntry=78"`. This link can be used to get [ABITransmission](#) records (as objects) from SMS-Server. You need only to create [IABITransmissionManager](#) manager and call `GetBySourceLink()` method:

```
string link = "CustomsEntry=78";  
var mngr = ClientContext.ServicesFactory.GetManager<IABITransmissionManager>();  
List<ABITransmission> result = mngr.GetBySourceLink(link, true);  
// true means that SMS-system returns also a  
// "raw" ABI Customs format text  
// in ABIMessage property of ABI transmission  
// objects.
```

And also you can use this link to get a list of all related events:

```
string link = "CustomsEntry=78";  
var mngrEvents = ClientContext.ServicesFactory.GetManager<SMS.Broker.ServiceContracts.Common.IEntityEventManager>();  
List<EntityEvent> Events = mngrEvents.GetABIEvents(link, true);  
// true means that SMS-system returns also a  
// "user-oriented" text in Text property of event  
// objects.
```

This approach (create a new object, save it, put to queue as Customs application) is general for all document (or directory) class objects those suggest data exchange with Customs (contact, Customs entry, ISF, FTZ admission, in-bond and others).

Detailed description of [ABITransmission](#) class and [IABITransmissionManager](#) interface you can see in 07. ABI classes and interfaces.pdf, 2. ABI Transmission.



### 5.3 ABI queries.

Another important part of data exchange with Customs is an ABI Query messaging. Please, pay attention that “ABI Query” at this context doesn’t mean a “query” in a general meaning but it means here one of Customs applications enumerated in CATAIR exactly as “Query” (Cargo Manifest / In-Bond / Entry Status Query, AD/CVD Query, Census Warning Query, ERF Queries and so on).

The difference from the previous approach (described in 5.2 *Documents and directories*) is that in ABI queries case you initially haven’t an object (document or directory) but only a set of parameters for put to queue methods and `ABIQuery` class object is created as a *result* of a put to queue operation.

For example, you want to send AMS Query (it’s Customs name is “Cargo Manifest / In-Bond / Entry Status Query”) for Bill of Lading. You need to create an `ISmsABIManager` interface object and run special `PutToQueue_CargoManifestEntryReleaseQuery_Bill()` method. This put to queue method creates outgoing `ABITransmission` class object (the same as in the previous approach) but in addition creates an `ABIQuery` class object. This last object plays the same role as a document or directory object (outgoing and incoming `ABITransmission` objects have a link to it as a source, for example, “ABIQuery=257”) but at this step you don’t know its link because `ABIQuery` object was created “off-screen”.

To simplify an access to a new created `ABIQuery` object you can add a `string` `clientRef` parameter to put to queue method parameters:

```
var mngrSmsABI = SMS.Broker.ClientContext.ServicesFactory.GetManager<SMS.Broker.ServiceContracts.ISmsABIManager>();
long ABITransmissionId =
mngrSmsABI.PutToQueue_CargoManifestEntryReleaseQuery_Bill(SCAC, BillNumber, requestRelatedBOL, clientRef);
where
string SCAC           - Bill of Lading issuer code,
string BillNumber     - Bill of Lading Number,
bool requestRelatedBOL - indicator, if true Customs returns related House or Master Bills information,
string clientRef      - Any string value that uniquely identifies a user or his request. For example, user phone number.
```

Then to get the `ABIQuery` objects list by `clientRef` you can apply `GetListByClientRef()` method of `IABIQueryManager`:

```
var mngrABIQuery =
SMS.Broker.ClientContext.ServicesFactory.GetManager<SMS.Broker.ServiceContracts.Documents.IABIQueryManager>();
List<ABIQuery> abiQueris = mngrABIQuery.GetListByClientRef(clientRef);
```

And as we already wrote above, when you know an object (now it is `ABIQuery` object) you can use its link to find related outgoing and incoming ABI transmissions. Incoming ABI transmission contains Customs reply to your query.

```
string link = "ABIQuery=257";
var mngrABITransmission = ClientContext.ServicesFactory.GetManager<IABITransmissionManager>();
List<ABITransmission> result = mngrABITransmission.GetBySourceLink(link, true); // true means that SMS-system returns also a
// "raw" ABI Customs format text
// in ABIMessage property of ABI transmission
// objects.
```

And also you can use known link to get all related events:

```
string link = "ABIQuery=257";
var mngrEvents = ClientContext.ServicesFactory.GetManager<SMS.Broker.ServiceContracts.Common.IEntityEventManager>();
List<EntityEvent> Events = mngrEvents.GetABIEvents(link, true); // true means that SMS-system returns also a
// "user oriented" text in Text property of event
// objects.
```

Another way to work with ABI Queries (without `clientRef` parameter) is to use returned `long` `ABITransmissionId` of synchronous put to queue method. It can be applied to get a new created `ABIQuery` object and Customs response.

More details about how to get `ABIQuery` object and response `ABITransmission` class objects by ABI transmission Id please look at 08. *Annex. ABI classes and interfaces.pdf*, 1.1 *AMS query samples, Sample 3*.

Detailed description of `ABIQuery` class you can see in 07. *ABI classes and interfaces.pdf*, 4. *ABI Query*.



## 6. Data Validation.

### 6.1 Document and directory exceptions handling.

Directory and document class objects include a lot of properties and collections. US Customs places high demands to data consistency of transmitting messages. The one and the same object (for example, [Contact](#) object) can be a source for a different type of messages to be sent to Customs (Importer/Consignee create/update, Importer/Bond query, Add Manufacturer). And different message types suggest different mandatory properties and collections.

To make a work with document and directory objects more easy for user a validation in SMS system is applied on two cases: validation on save operation (low requirements) and validation on put to queue operation (complete requirements). For example, a save process of Importer Security Filing document has only 3 validated parameters: Importer, TransportationMode (it must be Sea 10 or 11) and Carrier. Other parameters are validated later on put to queue operation.

To catch exceptions that appear while document and directory class objects validation is running we use [EntryValidationFault](#) class:

```
try
{
    newCE = mngrCE.Save(newCE);
}
catch (System.ServiceModel.FaultException<SMS.Broker.Faults.EntryValidationFault> ex)
{
    // Your some catch handling, for example a record to log file.
    log.Info(ErrorMessage(ex));
}
```

After a catching of exception ex.Detail property contains a reference to [EntryValidationFault](#) class object with validated object EntityLink, properties and related errors. A sample of its processing (how to get readable messages from it) you can see in *02. Annex. Getting Started With SMS API.pdf*, 6. Error message processing method.

*Note 5. FaultException<Tdetail>.*  
*It is a standard exception. Namespace: System.ServiceModel*

The same approach is used in the case of synchronous put to queue operation. Here we send to Customs CBPF 5106 (add a new importer):

```
try
{
    mngrContact.PutToQUE(ImporterId, Contact.ActionQUE.ImporterConsigneeCreate);
}
catch (System.ServiceModel.FaultException<SMS.Broker.Faults.EntryValidationFault> ex)
{
    // Your some catch handling, for example a record to log file.
    log.Info(ErrorMessage(ex));
}
```

You also can run a validation process “by itself” without execution of save or put to queue operations (we want to check Importer Security Filing (SF) application submitting before put it to queue):

```
try
{
    mngrISF.ValidateTreeById(isfld, "QUE SF");
}
catch (System.ServiceModel.FaultException<SMS.Broker.Faults.EntryValidationFault> ex)
{
    // Your some catch handling, for example a record to log file.
    log.Info(ErrorMessage(ex));
}
```



On the case of asynchronous put to queue method (`PutToQUEUEOneWay()`) the catch block can't get a validation object. But we can see the result of operation by another way. Each last status record (See *Note 2. LastStatuses. above*) on error case contains validation result of related operation. The result is placed to last status object's Text property. Its text is a serialized `EntryValidationFault` object that can be deserialized and used. Please see *02. Annex. Getting Started With SMS API.pdf, 7. How to get a validation result of asynchronous sent*. Detailed description of `EntryValidationFault` class you can find in *05. Common usage classes and interfaces. Part 1.pdf, 9.2 Entry Validation Fault*.

## 6.2 ABI query exceptions handling.

As it was explained above in *5.3 ABI queries* part on ABI queries case you initially have no an object (document or directory) but only a set of parameters for put to queue methods and `ABIQuery` class object is created as a *result* of a put to queue operation. Respectively, at the time of validation process that takes place during put to queue operation we have no an object which we could relate appeared errors to. `ABIQuery` class object is not created yet at this point. To solve this problem `ActionFaults` class is used. It doesn't require an object to be related to and returns only a list of incorrect parameters.

Here we run put to queue ABI Query method for carriers in a catch block:

```
var mngrSmsABI = SMS.Broker.ClientContext.ServicesFactory.GetManager<SMS.Broker.ServiceContracts.ISmsABIManager>();
try
{
    mngrSmsABI.PutToQueue_ExtractReferenceFile_Carriers();
}
catch (System.ServiceModel.FaultException<SMS.Broker.Faults.ActionFaults> ex)
{
    // Your some catch handling, for example a record to log file.
    log.Info(ex.Detail.ToString());
}
```

Detailed description of `ActionFaults` class you can see in *05. Common usage classes and interfaces. Part 1.pdf, 9.1 Action Faults*.

## 6.3 Possible values.

Data exchange with Customs suggests a usage of standardized codes (for units of measure, duty and fee classes, currency, country codes and many many others). Any document that contains incorrect codes is rejected by Customs ABI system. Possible values lists were added to SMS-system to avoid such rejections. They are used by SMS-system business logic for validation and also can be used on a client side.

The list consists of `PossibleValue` structures. Each possible value structure has two fields: Value and Description. For example: `"CM2"` and `"Square Centimeters"`. The name of a list of possible values for some property is constructed from concatenation of this property name and word "Value".

For example, in a table that describes `ShipmentArticleTariff` class properties you see that property `UnitOfMeasure1` has a note "Has Property Values" in "Description" column. It means that available values of this property are restricted to values of some possible value list. This list name is `UnitOfMeasure1Values` (`"UnitOfMeasure1"` + `"Values"`) and it is included to `ShipmentArticleTariff` class as a static property. You can apply to it:

```
List<PossibleValue> UOMs = ShipmentArticleTariff.UnitOfMeasure1Values;
```

and then use the UOMs list in user interface giving to user only eligible UOMs to choose.