##### 4-Tell Boost Extractor Specification

**Version 5.0**



Table Of Contents

[Overview - 5 -](#_Toc435596032)

[Personalized Product Recommendations - 5 -](#_Toc435596033)

[This Document - 7 -](#_Toc435596034)

[The Extractor - 8 -](#_Toc435596035)

[Integration Overview - 10 -](#_Toc435596036)

[Extractor Requirements - 11 -](#_Toc435596037)

[Choosing the Right Implementation - 11 -](#_Toc435596038)

[Installed Plugin Implementation Requirements - 12 -](#_Toc435596039)

[Admin Settings Page - 13 -](#_Toc435596040)

[Data Feed Definition - 15 -](#_Toc435596041)

[Data Feed Response Format - 18 -](#_Toc435596042)

[Examples - 22 -](#_Toc435596043)

[Test Scenarios - 24 -](#_Toc435596044)

[Direct API Implementation Requirements - 25 -](#_Toc435596045)

[Code Reuse - 25 -](#_Toc435596046)

[Extraction Routines - 31 -](#_Toc435596047)

[Overriding GetFeedData() - 33 -](#_Toc435596048)

[GetEntity() - 34 -](#_Toc435596049)

[Additional Base Class Overrides - 34 -](#_Toc435596050)

[Extraction Routine Output Format - 36 -](#_Toc435596051)

[TXT File Output Format - 37 -](#_Toc435596052)

[Building a Direct API Extractor – Step by Step - 37 -](#_Toc435596053)

[Extractor Performance - 43 -](#_Toc435596054)

[Testing an Extractor - 45 -](#_Toc435596055)

[Test Tool Operation - 45 -](#_Toc435596056)

[Supporting Installed Plugins - 49 -](#_Toc435596057)

[JavaScript Insertion - 50 -](#_Toc435596058)

[JavaScript for All Pages - 50 -](#_Toc435596059)

[Order Confirmation - 50 -](#_Toc435596060)

[Multi-Store & Multi-Contry - 51 -](#_Toc435596061)

[Multiple Stores - 51 -](#_Toc435596062)

[Multiple Countries (FUTURE) - 51 -](#_Toc435596063)

[Contact Information - 52 -](#_Toc435596064)

# Overview

## Personalized Product Recommendations

The 4-Tell Boost™ Recommendation Service uses a retailer’s catalog (including categories, brands and price), sales and click stream to produce and display product recommendations (‘People also bought…’) on his website.

The product recommendations are generated and displayed on each page view, and change based on the actions of the shopper (e.g. recently viewed products, items in the cart and past purchases).

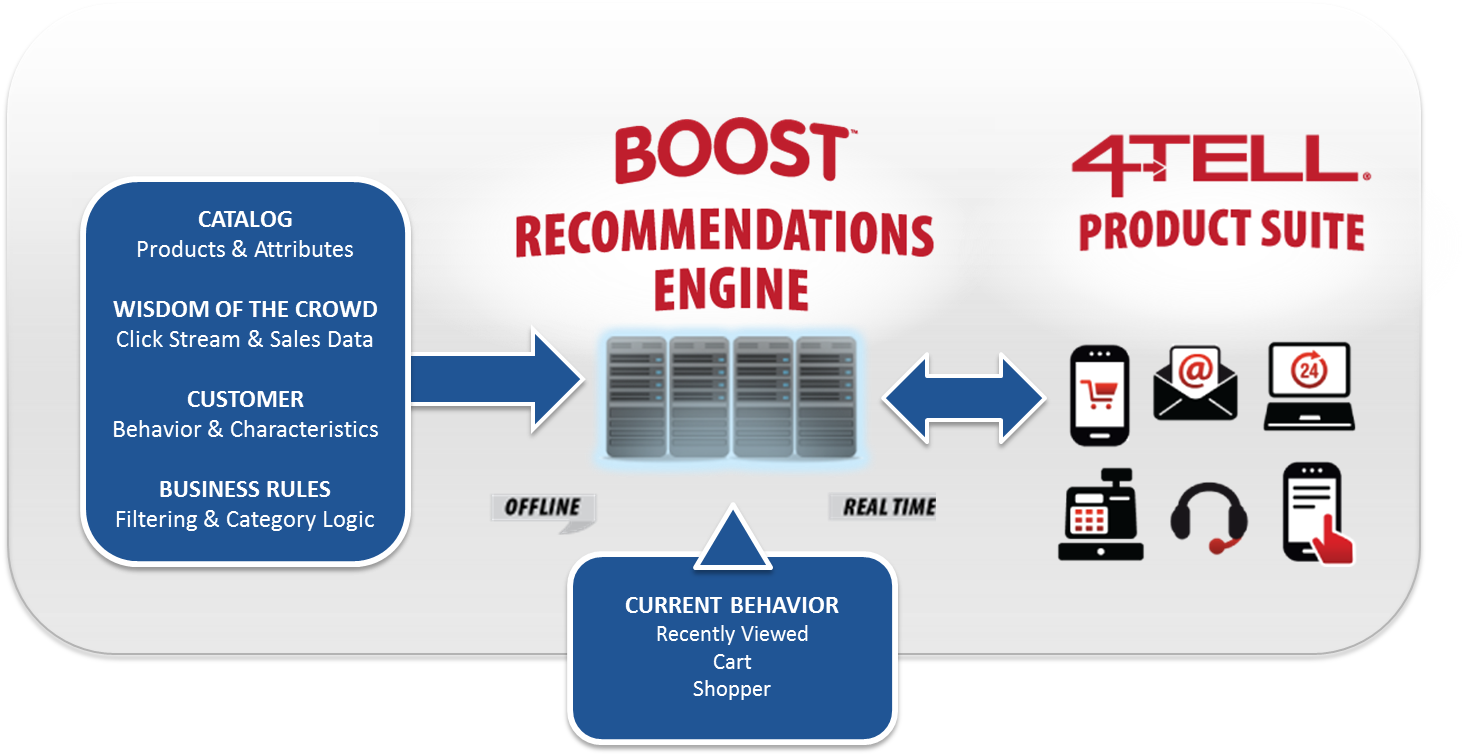
The base recommendation types that we display are:

1. Cross-sell (e.g. pants are bought with a shirt)
2. Similar or upsell (e.g. other shirts you may like when looking at a shirt)
3. Top sellers (site wide or for a category)
4. Personalized (e.g. the products that each customer is likely to buy).

These base recommendation types correspond to the base recommendation lookup tables that are created nightly by the Boost Generator Engine. The engine uses the catalog, 18 months of sales, and 10 weeks of click stream data when generating the tables. It gets some of this information using the “**Extractor**” that this guide describes.

The tables are then used by our recommendation service when dynamically determining which recommendations to display on each page that a shopper visits. Our recommendation delivery engine takes into account a customer’s past purchases, recent page views, and the items they may have already placed in their cart.

The recommendations are served up by our web service via a RESTful API and displayed on the retailer’s website via JavaScript that is inserted into each page by 4-Tell.



To start the process we initially extract the retailer’s entire catalog and historical sales data. From then on, we get nightly catalog updates, and collect real-time sales and click stream data. All of this is used to generate the recommendations we need.

## This Document

This guild defines the requirements for the “Extractor” that must be built to interface the 4-Tell Boost™ Recommendation Service to an existing Ecommerce Platform.

The Extractor is responsible for gathering the “static” part of the information needed to generate the recommendations that we display on a retailer’s site.

To understand where the Extractor fits in consider the following sequence of steps in the recommendation generation life cycle:

1. Sign up a site that wants 4-Tell Recommendations.
2. Define a site integration plan with the retailer that identifies where recommendations should be displayed and what type of recommendations should be displayed at each identified location.
3. Implement the site plan by adding JavaScript at each identified location that will both request recommendations from the 4-Tell Recommendation Service and display them.
4. Run the “Extractor” from our Dash Service to gather the initial (or “static”) catalog, sales and customer data needed by the Recommendation Service.
5. Use the extracted information to generate a base set of recommendation tables.
6. Take the site live.
7. Recognize customer interaction with the retailer’s site.
8. Supplement the base recommendation tables with real-time (or “dynamic”) customer click history and cart content information. This is done by post calls from the site to the 4-Tell Recommendation Service.
9. Request recommendations from our service based on the supplemental information and the base tables originally generated during extraction.
10. Display those recommendations on the page currently viewed by the customer.

Repeat steps 7 to 10 many times.

1. Periodically re-run the Extractor to update catalog, sales and customer data.

Note that steps 1-6 are generally done once, while steps 7-10 are done frequently and repeatedly as customers interact with the retailer’s site. Step 11 is scheduled to automatically update the base information kept within the service on a regular basis (usually nightly).

## The Extractor

The “Extractor” simply uses the API provided by an ecommerce platform to gather the static data we need from a retailer’s site and deliver it as a set of files in a known format to our service.

The basic set of data gathered during extraction is:

1. Catalog Data
2. Sales Data
3. Customer Data

Optionally the extractor gathers:

1. Inventory Data
2. Category Data
3. Manufacturer Data
4. Department Data

The “Extractor” for a given platform is comprised of one *or* two parts:

1. In the case where it has one part, the extractor is hosted by 4-Tell and does not affect (or is known by) the ecommerce server.

This is the most direct implementation of the Extractor, but is also the most intrusive to our Dash Service. In this case the extractor speaks directly to the ecommerce API (usually a RESTful API providing XML or JSON output) to pull the data needed and then write it to the files that will be consumed by our generator. These files are txt files written in a specific format. We call this a **Direct API Implementation**.

1. In the case where it has two parts, the extractor is broken into two sides that talk to one another, one hosted by 4-Tell and one hosted by the ecommerce platform or by the retailer’s site running the platform’s software.

This implementation introduces a “**Plugin**” that resides on the ecommerce platform server and is responsible for all interaction with the ecommerce platform API itself. The Plugin exposes a RESTful API (JSON) that adheres to the single general format defined below. In this way any platform can be integrated with our Dash Service (without modifying that code base) by supplying a Plugin that meets the interface specifications described in this document. In this case the side of the Extractor hosted by 4-Tell will not need to be modified. It will extract (based on the specified interface) from the new platform via the new Plugin for that platform. We call this an **Installed Plugin Implementation**. This is the preferred implementation because it isolates our system from the details of a specific platform and can be used to manage changes in a platform’s API without affecting our Dash Service code base. However, it is not always possible to introduce a Plugin on the retailer’s side of things, some platforms do not provide for script (application) hosting.

Note that depending on the platform, a “Plugin” may be referred to as a “Connector”, “Module” or “Extension”.

## Integration Overview

We call the process of building recommendations into a retailer’s site, “Integration”. There are several main steps to integrating a store with our service:

1. The first step is for the store owner to signup for the Boost service on our website at   
   [www.4-tell.com/signup](http://www.4-tell.com/signup). The store owner will then receive a unique Client Alias and a Service Key to identify their store to our service. They will also work with our Account Management Team to establish a site plan that will be sent to them for approval.
2. Upon approval our Integration Team will do the work needed to add our recommendations to the store site.
3. In the situation where the store’s ecommerce platform supports an Installed Plugin, the store owner will receive instructions on how to install our plugin. They will install our plugin and enter their Client Alias and Service Key into the module settings.
4. In the situation where the store’s ecommerce platform does not support an Installed Plugin, the store owner will enter all credentials needed by our Direct API Implementaion (usually an API key and API user name) into our Dashboard.
5. In either case, our Extractor will pull the catalog, sales, and customer data needed to build an initial set of base recommendations. These results will be immediately available in our Boost Dashboard® for review, and the store owner can make business rule adjustments to best match their own merchandizing practices. Recommendations will also be available in our Boost Web Service, waiting for calls from the retailer’s web pages and/or emails requesting recommendations to be displayed.
6. Integration will include the introduction of a small amount of JavaScript added to the store’s main site frame template. After the store owner approves the site plan, we will activate the display settings on our service.

When a shopper opens a retailer’s page, our JavaScript on that page will call our Boost Web Service to get recommendations, with different options requested depending on the information available for the user and page being viewed. Our web service will return the requested number of recommended products, and our javaScript on the page will display the recommendations.

# Extractor Requirements

## Choosing the Right Implementation

Answering a few questions should tell the Developer which extractor type to implement.

As stated above the preferred implementation is an Installed Plugin because it isolates our system from the details of a specific platform and can be used to manage changes in a Platform’s API without affecting our server side code. However, it is not always possible to introduce a Plugin on the retailer’s side of things, some platforms do not provide for module (application) hosting, so the questions to ask are:

1. Does the Platform host the retailer’s site?
2. If the answer to #1 is yes, then:
   1. Does the Platform also provide module hosting for 3rd-party plugins like ours?
   2. If the answer to #2a is yes, then the Developer should choose to create an Installed Plugin.
   3. If the answer to #2a is no, then the Developer should choose to create a Direct API Extractor.
3. If the answer to #1 was no then:
   1. The Platform must be hosted on a retailer’s server, so ask whether it’s possible to install a module on that server to integrate with the Platform?
   2. If the answer to #5 is yes, then the Developer should choose to create an Installed Plugin.
   3. If the answer to #5 was no, then the Developer should choose to create a Direct API Extractor.

Note that depending on the platform, “Module Hosting” may be synonymous to “the ability to add” a “Plugin”, “Connector”, “Module” or “Extension”.

The following two sections cover the requirements for the Installed Plugin and Direct API Implementations of an Extractor.

# Installed Plugin Implementation Requirements

The 4-Tell Boost Plugin will require four main capabilities:

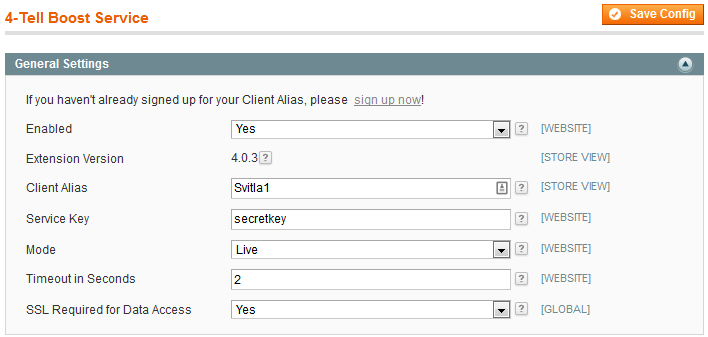
1. **Admin** **Settings Page** – Create an admin page to allow store owners to set plugin controls.
2. **Data Feed Definition** – Expose a URL that allows our service to request data from our plugin with query parameters and receive the data as a JSON formatted response.
3. **JavaScript Insertion** – Insert a small amount of JavaScript into the main frame header and order confirmation page template.
4. **Multi-Store & Multi-Country–** If the platform supports multiple stores corresponding to multiple countries and/or supports the display of different languages and currencies on the same store, our plug-in needs to support these options as well.

## Admin Settings Page

The 4-Tell Boost plugin will provide an admin page to the store so that the store owner can define their extraction settings:

1. Client Alias
2. Service / API Key
3. Mode (Live or Staging)
4. Image Size
5. Optional Loader Block
6. Other Platform Specific Parameters

As an example, the following image shows a section of one Admin Page.



### Client Alias Input

The **Client Alias** is the way that we identify a client. It is used in the web service call for recommendations. It is a string value with 8 characters or less. We provide the Client Alias to the client, and they enter it into the admin page.

Sometimes, if the platform creates an API token, the client alias is not needed in the admin panel, since the token is added to our site rules for our extractor to use as it pulls the catalog. This assumes that when one backend supports multiple stores, the token is different for each store (see Multi-Store/Lingual/Currency section). Otherwise, we need to enter a different client alias into each store such that the data feed only pulls data from that store.

### Service Key Input

The **Service Key** is the secret used to authenticate calls to the plugin. In other words, it makes the feed secure so that other’s cannot use the feed to extract valuable data. Furthermore, responses should also be blocked if the value is blank. The service key is a string value with a maximum size of 24 characters.

There are two options for the Service Key. The goal is to create secure process that is easy to setup and maintain.

1. If the platform creates a token when the API is enabled, that token should be used for the Service Key. If the token has several fields (e.g. client ID and API Key), we can add fields to our feed URL to match the field names for the platform – and not use the Service Key field. The token should be displayed on the Admin page. The client should be able to copy the token, but not edit it. This allows the client to provide the token to 4-Tell such as copying the token into our secure Dashboard.
2. If the platform does not create an API token, the Service Key is provided by 4-tell, and the client enters into the Service Key value on the admin page.

### Mode Input

The third setting is the **Mode**. This choice should be a drop-list selector that can either be set to “Live” or “Staging”. The Mode becomes a variable that our JavaScript can read from the web page.

### Image Size Input

The fourth setting is **Image Size**. This is the thumbnail image used in the recommendation display. Typically an image size is set or selected from a list, and the resulting link is used in the catalog feed (see below).

### Optional Loader Block Input

This is a desired, but optional requirement. Provide a textbox to allow entry of the Loader Block to be inserted in the retailer’s header template. There will need to be code behind to actually insert the block as well.

### Runtime Parameters

The ClientAlias and Mode values need to be made available as **Runtime Parameters** that can be accessed from JavaScript on the main frame of the site. See below for details on JavaScript Insertion.

## Data Feed Definition

The 4-Tell Boost Plugin will expose a URL that allows our recommendation service to request data with query parameters. All error reporting should use standard http error codes returned as response headers. Valid responses are formatted as JSON string arrays (see below for details).

### Security

Connection for this URL requires SSL and **Basic Authentication**. If Basic Authentication is not supported on the platform, please let us know and we’ll use a different authentication strategy.

**Non-https requests should elicit a “403 Forbidden” error response.**

**NOTE: Often, a client’s staging site will not have access to SSL certificates. For this reason, we allow an admin setting to disable the SSL requirement. When disabled, the client should be presented with a warning that explains that SSL protection is highly desired for all production sites.**

### Feed URL

The feed URL provides a new page on the store’s website that only our service can access. Calls from our service to this URL will include query parameters such as the secret keys for security, the data type requested, and parameters to filter the results that are returned.

Request parameters will include (default values in bold):

1. ClientAlias and ServiceKey (Security Credentials)
2. DataGroup (“**Catalog”**, “CategoryNames”, “ManufacturerNames”, “Customers”, “Sales”, “Inventory”, “Returns” or “Version”)
3. ResultType ( “**Data**”, “Count” )
4. CountryIndex (“**1**”, any integer up to the number of countries for the store)
5. DateRange (Default is all dates)
6. RowRange (Default is all rows)
7. ExtraFields (Optional)

### Client Alias and Service Key

The **ClientAlias** and **ServiceKey** are the security credentials (formatted as strings). We'd prefer to send these as a basic authorization header (so they are not saved in logs), but can also send them as query parameters if basic authentication is not supported.

The ClientAlias identifies the client and is sent in the recommendation requests. The ServiceKey is the secret to enable the API for the feed.

Our plugin should have an admin configuration page where these values can be entered by the store owner after they sign up for our service. If there is a mismatch or if either of these admin settings are blank, the module should return a “401 Unauthorized” error when feed data is requested.

### DataGroup

**DataGroup** determines which type of data is being requested. Valid entries are “Catalog”, “Sales”, “CategoryNames”, “ManufacturerNames”, “Customers”, “Inventory”, “Returns” and “Version”. The default value is “Catalog” if no DataGroup is provided.

The format for every group except “Version” is provided in the next section (Data Feed Response Format). “Version” requires no other query parameters and returns the version of both our plugin and the cart platform. If the cart is not a hosted platform, it should also return the OS type, OS version, number of processors, and amount of RAM. The results should be formatted as a JSON array of tag-value pairs. The results for the other DataGroup choices are defined by the additional parameters below.

### CountryIndex

**CountryIndex** is only included for stores that have multiple currencies and languages for multiple countries. It is usually not included, and defaults to 1.

If included, it determines the index of the country to pull the catalog data. It only applies to the catalog data. It assumes that the platform includes an index for each country (see Multi-Country section below for more details).

### ResultType

**ResultType** determines whether the results are the actual data requested or just a count of how many rows of data would be provided for the given set of query parameters. Valid entries are “Data” or “Count”. “Data” is the default and used if no “ResultType” is provided. For “Data”, the format of the results is provided below in the Sales, Catalog, Category Names, Manufacturer Names and Customer sections. For “Count”, the result is an integer representing the number of data rows in the results. It is used by our service to gauge whether it will be necessary to segment the calls for data so that each set of results is not too large.

### DateRange

**DateRange** specifies the beginning and end dates for results. If “DateRange” is not provided, all dates must be included in the results. Date ranges are used to create updates of catalog, sales, and customers. The range is typically one month, but can be any number of days. The dates will be entered as two dates separated by a comma (i.e. DateRange=2013-08-01,2013-08-31). If no end date is included, it is assumed to be the current date. The range includes actions on the beginning and end dates (i.e. **inclusive of the end date**).

Date range has different meanings for each data group:

* For the catalog, the product must have been created or modified in this date range.
* For sales data, the sale must have occurred within this range – which means that sales uses the created date only. It does not use the modified date; otherwise, the sale is duplicated when shipped.
* For customer data, the customer must have bought something, created or modified their account in this date range. For some platforms, the updated\_at field tracks both creating and updating the customer profile.

### RowRange

**RowRange** specifies the first result row and maximum number of rows to return. This allows our service to segment the data from a particular query so that we only receive a subset of the results. The first catalog entry is RowRange=1 (not 0 indexed). If the second parameter is 0, only the header should be returned. If “RowRange” is not provided, all rows are returned.

For example: "RowRange=1,5000" would return the first 5000 rows of the results. “RowRange=5001,5000” would return the next 5000 rows, which is row 5001 to 10,000.

It is important that all internal data calls or database queries are optimized to only pull the rows requested. In other words, the database query should not request the entire catalog or the entire sales history, and then feed only those items in the row range requested. Programming the call in this manner would cause memory issues for large data sets.

### ExtraFields

**ExtraFields** specifies additional attributes to include in the catalog export or customer export. These fields are returned in addition to the standard product or customer fields. See below for the standard list of fields to return. It is important to anticipate additional fields that might be desired and make sure that our feed can provide those values, even if they are stored in a separate table (such as custom product prices or attributes).

Any requests that do not conform to the parameters above should cause a “400 Bad Request” error.

## Data Feed Response Format

Feed data is used to generate the Boost recommendation tables. Calls from our service to the plugin URL discussed above must return the data requested according to the following requirements. All results are to be formatted as JSON string arrays, where the first row of the array lists the column headers and the rest of the rows only hold the values.

### Version Feed Response

The version response is a list of paired items, including the relevant entries from the list in the following example. Many plugins are hosted and only return the plugin and platform version. **The Countries field is important since it tells our system how many catalogs to pull.**

{"Plugin":"4.0.7","Platform":"1.7.0.2","Countries":"1","Memory":134217728,"OSType":"Linux","OSVersion":"#64-Ubuntu SMP Mon Mar 25 21:42:18 UTC 2013"}

### Catalog Feed Response

The Catalog request should return one row for each product (and variant of that product) in the store catalog. The header row determines the order of the data fields exported. The field names listed below are the standard fields, e.g. ["ProductID", "Name", "CategoryIDs", "ManufacturerID", "Price", "SalePrice", "ListPrice", "Cost", "Inventory", "Visible", "Link", "ImageLink", "Rating", "StandardCode", "ParentID"]. Not all fields are required and they can be listed in any order. Additional fields may be requested by using the query parameter “&ExtraFields=field1,field2,field3” where the field names would exactly match fields in the customer’s product table.

**Note**: All products should be listed regardless of stock or active settings. This is so that our solution can learn from the sales of the product to improve recommendations for other items. The store owner will set exclusion rules in our dashboard to make sure discontinued or out of stock items are not recommended.

**ProductID**: The Product ID is the unique product identifier that the ecommerce platform uses to identify the product. The SKU, UPC or other standard ID is included in the standard code field. **If the product is a child (e.g. size or color), the parent ID is included in the ParentID field (listed below). You CANNOT include commas within the Product ID.**

**Name:** The Name is the display name of the product, and can include any standard characters.

**CategoryIDs**: The Category IDs parameter is multidimensional, meaning that a product can belong to several categories. When a product belongs to multiple categories, the IDs are separated by commas (,). The most important category is preferably listed first, as this will be the default category used in our dashboard display. However, this order does not affect recommendations. You CANNOT include commas within the category IDs. The category ID can be any number or alpha-numeric ID.

**ManufacturerID:** The manufacturer or brand id can be numeric or alphanumeric. This field can also hold the actual name if IDs are not used. If it is not available, it should contain an empty string.

**Price**: The Price is the product price as a decimal value without the currency symbol.

**SalePrice**: The Sale Price is also a decimal value without the currency symbol. It should only contain a value if the item is currently on sale. Otherwise, it should contain an empty string.

**ListPrice:** The manufacturer suggested retail price (MSRP). If it is not available, it should contain an empty string.

**Cost**: The cost of the product. If it is not available, it should contain an empty string.

**Inventory**: If the store actively tracks inventory, then this field should show the number of items available for this product. If the store does not track inventory this parameter should contain an empty string.

**Visible**: This is the flag that tracks whether or not an item is active in the store. 1 = visible, 0 = not visible.

**Link**: This is the link to the product page for this item. This link includes the full path and starts with ‘http://’.

**ImageLink**: This is the link to the thumbnail image that should display with the recommendation, preferably a cached image so it loads fast. It is usually the same thumbnail image that is used with category or search pages. This link includes the full path and starts with ‘//’ (without the ‘http:’).

**Rating:** This is the product rating as a number, usually between 1 and 5. If no rating exists, the field should be blank and not have a 0.

**StandardCode**: This is the standard product code for the product. It can be the UPC, ISBN, or any other standard code. Even if the product ID is the same code, please include the code in this field again.

**ParentID:** If the product is a child product, provide the parent ID for the product in this field. Parent products should leave this column blank and return an empty string instead.

**Extra Fields:** Extra catalog fields can be requested using the “ExtraFields” option in the URL query string. This list can include any product attribute that the platform contains, including custom attributes. For example, the retailer may have a field for premium versus sale items, or shipping options (direct versus drop) or alternate prices. We’ve even had customers who created a ‘4Tell’ attribute that provided desired information for each item. Where needed, the plugin should define special names or even compound names that our extractor can request to access specific data in other related tables on the platform (i.e. "CustomFields.1" to access the first custom field for an item in a separate "CustomFields" table).

### Inventory Feed Response

The header should be [“ProductID”, “Inventory”] and the inventory for items that have changed in the specified data range, or for all items if no date range is specified, is returned. This will allow us to update the inventory we have in our image of the retailer’s catalog without pulling the entire catalog.

### Category Names Feed Response

The CategoryNames response contains one record for each category ID followed by its display name. The format is [“ID”, “Name”]. This assumes the platform uses category IDs in the catalog in place of their actual names. If the names are listed directly in the catalog, this file is not necessary and the response to this request can be blank.

### Manufacturer Names Feed Response

The ManufacturerNames response contain one record for each manufacturer or brand ID followed by its display name. The format is [“ID”, “Name”]. This assumes the platform uses manufacturer or brand IDs in the catalog in place of their actual names. **If the names are listed directly in the catalog, this file is not necessary** and the response to this request can be blank.

### Customer Feed Response

The Customer response contains one record for each customer who has either:

* Bought something OR
* Created a profile OR
* Changed their profile during the date range.

Often the customer update date includes both new profiles and profile creation. Typically, this is merged with a JOIN SQL query for customers with orders during the date range.

The fields contain [“CustomerID”, “Email”, “Name”, “Address”, “City”, “State”, “PostalCode”, “Country”, “Phone”, “Gender”, “Birthday”, “AgeRange”, “AlternativeIDs”, “DoNotTrack”].

The required fields are:

* **CustomerID** is the ID for the customer
* **Email** is the email address for the customer
* **Name** is the customer’s full name.
* **State** is the state or province for the customer
* **PostalCode** is the zip or postal code for the customer

Optional fields:

* **Address** is the street address, including address lines 1 and 2 (and 3 if included)
* **City** is the city for the customer
* **Country** is the country name for the customer
* **Phone** is the customer’s phone number
* **Gender** is the gender of the customer and can be skipped if not available
* **Birthday** or **AgeRange** are the birthday or age range of the customer, if available
* **AlternativeIDs** are useful if the store has additional customer IDs for additional platforms that will use recommendations, such as in-store POS or ERP. If not, leave this field blank.
* **DoNotTrack** is a flag that says the customer has elected to not be tracked. We save no information for this customer. 0 = Yes, okay to track. 1 = Do not track. A blank entry assumes that it is okay to track.

Additional database fields can be requested through the ‘ExtraFields’ parameter. For these fields, the database field name is known and, if included, should be returned by the feed.

### Sales Feed Response

The sales feed should only include the sales that were:

* Created during the date range

And not updated in the date range since that will duplicate sales when they are shipped.

Each sales record consists of an [“OrderID”, “ProductID”, “CustomerID”, “Quantity”, and “Date”], with one record for each sale. Repeat sales (i.e. same Product ID and Customer ID) are important and should not be ignored. **Returns are also important, and should have a negative quantity.** Returns can be requested by themselves (see Returns sub-section below).

### Returns Feed Response

This feed is required to offset the real-time sales via order confirmations sent to our API. This call should use the same header as sales with negative quantity for the returns. Specifically, the result is an array where each row contains a [“OrderID”, “ProductID”, “CustomerID”, “Quantity”, and “Date”], and quantity uses a negative value (i.e. -1 if one of the product is returned, -2 if two are returned, and so on). These results are provided for all returns during the date range (or all returns that exist if no date range is provided).

## Examples

### Example Feed URL Requests

https://www.siteX.com/4TellFeed?DataGroup=Catalog&ClientAlias=SiteX&ServiceKey=122fb552-3ed6-4c24-ace9-cb41d5639921

https://www.siteX.com/4TellFeed?DataGroup=Sales&DateRange=2013-08-01,2012-08-31& ClientAlias=SiteX&ServiceKey=122fb552-3ed6-4c24-ace9-cb41d5639921

https://www.siteX.com/4TellFeed?DataGroup=Customers&DateRange=2013-08-01,2012-08-31&ClientAlias=SiteX&ServiceKey=122fb552-3ed6-4c24-ace9-cb41d5639921& ExtraFields=Name,Address,City,State,Country,Phone,Gender,AgeRange,AlternativeIDs, DoNotTrack

For Basic Authentication, the Client Alias and ServiceKey are sent in the authentication header rather than the URL request.

If you’d rather have a separate URL for each data group instead of using the DataGroup parameter, we can do that. Just let us know.

### Example Data Responses

The returned data is shown below for the two example calls. The data response is JSON format. In order to further reduce message size, please use string arrays and include the data header tags as the first row rather than repeating the tags with each data row. Finally, header tags should not contain any spaces.

A response with two products in the catalog would look like this (where Gender was requested as an extra field):

[["ProductID", "Name", "CategoryIDs", "ManufacturerID", "Price", "SalePrice", "ListPrice", "Cost", "Inventory", "Visible", "Link", "ImageLink", "Rating", "StandardCode", "Gender"],

["A31", "Men's Assault Spring™", "22, 24,33", "32", "2", "1.8", "2", "1", "1000", "True", "//siteX.com/PID?31", "//siteX.com/image/t31", "2", "A31-ABC", “Men”],

["A32", "Men's Epic 2 4/3 Full", "22,35,78", "33", "3", "2.6", "3", "1.5", "789", "False", "//siteX.com/PID?32", "//siteX.com/image/t32", "3", "A32-ABD", “Men”]]

A response with three sales records would look like:

[[“OrderID”,”ProductID”,”CustomerID”,”Quantity”,”Date”],  
[“1”, “hsfPBcasCarm”,”61DE45FC1”,”2”,”2013-08-01”],  
[“1”,”synSamplePack”,”6007DF77”,”1”,”2013-08-01”],

[“2”,”synBasePad”,”6117DF77”,”1”,”2013-08-02”]]

A response for three customers would be:

[[“CustomerID”, “Email”, “Name”, “Address”, “City”, “State”, “Country”, “Phone”, “Gender”, “AgeRange”, “AlternativeIDs”, “DoNotTrack”],

["12","ken@4-Tell.com","Ken Levy","110 NE Cedar", "Stevenson", "WA", "United States", "509.427.5374","Male","40-50","","0"],

["12","bobw@gmail.com","Bob White ","22 White Street, Suite 250", "Buffalo", "NY", "United States", "716.227.5274","Male","40-50","","0"],

["15","joseysmith1@yahoo.com","Josey Smith ","1 Main St", "Portland", "OR", "United States", "503-516-1800","Female","20-30","","0"]]

The order of the header row specifies the order of data in each following rows. All rows must have the same number of elements and match the order of the header row. **If a particular row does not have a value for one of the fields, that field should be listed as an empty string.**

Note that The IDs can be numeric or alpha-numeric. All dates can either be in mm-dd-yyyy or yyyy-mm-dd format (using either dashes or forward slashes).

## Test Scenarios

We test the plugin with the following steps:

* Compare catalog items in platform’s admin panel to the full feed to the catalog.txt file that is saved on our server.
* Compare sales in the platform’s admin panel to all monthly sales saved on our server.
* Compare customers in the platform’s admin panel to feed of all customers to monthly customer files saved on our server.
* Compare category names in the platform’s admin panel to feed data to file saved on our server.
* If manufacturer names are not sent directly in the catalog, compare manufacturer names in the platform’s admin panel to feed data to file saved on our server.
* Test Version, and verify the results.
* Test Count and RowRange for (i) catalog, (ii) sales for one month, and (iii) customers for one month.
* Test Inventory by changing two items, pull feed for current month, and see if included.

# Direct API Implementation Requirements

When the Extractor is a Direct API Implementation the requirements are narrowed to what must be implemented within the 4-Tell Dash Service to extract data using the ecommerce platform API. Within the Dash Service a set of extractors exists, one for each supported Direct API platform. The current list includes the following and is growing:

* 3dcart
* AspDotNetStorefront
* BigCommerce
* CommerceV3
* Custom site (Do It YourSelf)
* Magento
* Miva Merchant
* NetSuite
* OS Commerce
* Shopify
* Volusion
* Others

## Code Reuse

Much of the functionality of an extractor is independent of the actual API used to pull the data needed by our system. Because of that a good portion of the code behind an extractor can be reused and does not need to be re-implemented by the Developer. This makes things easier when bringing up a new extractor and makes our system of extractors more compact and understandable.

The areas of functionality that are common to all extractors and do not need to be re-implemented for a new extractor are listed below (the responsibilities of the Software Developer are highlighted in green in this section):

1. **Site Rules Handling**. The Site Rules are a big part of our system. They are kept in a file called “SiteRules.xml”, one file for each store. An example is shown here. (***Note that the arrows in this example flag values that are explained in the paragraph immediately following the example***).

<?xml version="1.0" encoding="UTF-8"?>

[<siteRules>](file:///C:\\Users\\Zac\\AppData\\Local\\Temp\\SiteRules.xml)

<alias>**testsite**</alias>

<cartType>**BigCommerce**</cartType>

<updateTimer hourOfDay="**18**" rate="**Daily**" extractType="**Update**" enabled="**true**"/>

<lastExtractionType>**Sales**</lastExtractionType>

<lastExtractionTime>**2015/10/29T01:28:49.329706807:00**</lastExtractionTime>

<apiUrl><https://www.testsite.com/api/v2/></apiUrl>

<apiUserName>**4tell**</apiUserName>

<apiKey>**e0daafe159b9c20d2ab4244af00b5**</apiKey>

[<extractorCredentials>](file:///C:\\Users\\Zac\\AppData\\Local\\Temp\\SiteRules.xml)

<type>**BasicAuth**</type>

<userName>**4Tell**</userName>

<password>**e0daafe159b9c20d2ab4244af00b5**</password>

<requireSsl>**true**</requireSsl>

</extractorCredentials>

<apiAcceptHeader>application/json</apiAcceptHeader>

[<cartWebClientConfig>](file:///C:\\Users\\Zac\\AppData\\Local\\Temp\\SiteRules.xml)

<sessionTimeout>**600**</sessionTimeout>

<responseTimeout>**600**</responseTimeout>

<retryDelay>**2000**</retryDelay>

<maxTries>**3**</maxTries>

<connectionLimit>**48**</connectionLimit>

<securityProtocol>**Ssl3,Tls**</securityProtocol>

<protocolVersion>**1.1**</protocolVersion>

<keepAlive>**false**</keepAlive>

<expect100Continue>**false**</expect100Continue>

<allowUnsafeHeaderParsing>**false**</allowUnsafeHeaderParsing>

</cartWebClientConfig>

<orderIdField>**id**</orderIdField>

<orderProductIdField>**product\_id**</orderProductIdField>

<orderCustomerIdField>**customer\_id**</orderCustomerIdField>

<orderQuantityField>**quantity**</orderQuantityField>

<orderDateField>**date\_created**</orderDateField>

<orderDateReversed>**false**</orderDateReversed>

<attribute1 fieldName="**categories**" name="**Category**" enabled="**true**"/>

<attribute2 fieldName="**brand\_id**" name="**Brand**" enabled="**true**"/>

<imageLinkBaseUrl>/**product\_images**</imageLinkBaseUrl>

<rulesEnabled>**Exclusions**</rulesEnabled>

[<exclusionConditions>](file:///C:\\Users\\Zac\\AppData\\Local\\Temp\\SiteRules.xml)

<condition fieldName="**is\_visible**" value="**false**" comparison="**Eq**" name="**Not Visible**"/>

<condition fieldName="**inventory\_level**" value="**1**" comparison="**Lt**" name="**Out Of Stock**"/>

<condition fieldName="**price**" value="**.01**" comparison="**Lt**" name="**Free**"/>

</exclusionConditions>

[<exclusionStats>](file:///C:\\Users\\Zac\\AppData\\Local\\Temp\\SiteRules.xml)

<stat value="**324**" name="**Not Visible**"/>

<stat value="**538**" name="**Out Of Stock**"/>

<stat value="**7**" name="**Missing Image**"/>

</exclusionStats>

</siteRules>

Relative to extraction for a specific site, the Site Rules keep the following information (highlighted by arrows above according to the sequence below):

* 1. INPUT: Site alias.
  2. INPUT: Site ecommerce platform type.
  3. INPUT: The extraction schedule and type of extraction to be made.
  4. OUTPUT: The last extraction data and type.
  5. INPUT: The credentials needed to access the platform API. Includes the base URL of the platform API, the site specific user name and key.
  6. INPUT/OUTPUT: The “extractorCredentials” section is automatically generated by the extractor if it is not specified here. It provides the authentication type (which defaults to BasicAuth), credential details (which depend on the authentication type), and SSL requirement.
  7. INPUT: “apiAcceptHeader” specifies whether the API is Json or XML.
  8. INPUT: Parameters controlling the web client configuration.
  9. INPUT: The names of required fields for sales data within the API.
  10. INPUT: The names of the categories and manufacturer (brand) within the API.
  11. INPUT: The image link base used by the API for product images in the Catalog.
  12. INPUT: Exclusion rules for products in the catalog.
  13. OUTPUT: Exclusion results during last extraction.

The 4-Tell Integration Specialist will be responsible for modifying many of the site rules for a site during the Integration Process. He does that according to the requirements of the site and relative to the ecommerce platform type of the site.

The Software Developer of a new extractor will provide an example SiteRules.xml that can be used as a template by the Integration Specialist when he brings up a site that is using the new extractor, i.e. one that has the same platform type as the extractor.

The Site Rules for a site are read in and written out during the extraction process by the base class of the extractor; (CartExtractorBase.cs). This handling is self-contained and not the responsibility of the Software Developer bringing up a new extractor.

What is the responsibility of the Software Developer is the utilization of the site rules themselves as he builds the extractor. By looking at an existing extractor that accesses an API similar to the one he is building, the Developer can see how the rules are used.

In addition to a site specific rules file there is a defaults file for each platform. For instance the following file holds the defaults for BigCommerce. This file can be found in “C:/ProgramData/4-Tell2.0/BigCommerceRules.xml”.



The Developer will need to create a file like this for the new extractor/platform.

1. **Http Web Client Handling**. The Site Rules provide the specifications for the client accessing the Platform API and they do need to be modified by the Developer accordingly, but the code that sets up the client and calls the API does not need to be re-implemented. Those setup and call routines exist in the base class and should be re-used in the new extractor. These calls are SetupDataFeed() and GetQueryResponse().
2. **URL Request Assembly**. GetQueryResponse() is the routine used to send a URL GET request to the API and get a JSON response back. This routine can be re-used as is (without modification by the Developer), but the URL request itself must be assembled specifically according to the Platform API and the request being made.
3. **Extraction Routines**. The job of an extractor does not change based on the platform it is accessing. Because of this, the structure within the extractor base class does not change for a given extractor and the calls needed to support our system are well defined and general. The Software Developer in this case only needs to override this set of calls, constructing the appropriate URLs, using GetQueryResponse() to post them to the client and manipulating the data response within the calls. This manipulation of data simply consists of parsing the data needed from the JSON response and formatting it as expected by the base extractor class.
4. **Final Processing**. The final output from an extractor is the set of txt files containing the extracted data in the format understood by the generator. This format is covered in the “Output Format Specification” section below. The data returned by the extraction routines is converted into Tables during this processing and then written out as txt files. The routines that do this are called “ProcessData()” and “WriteTable()” and are generic to all extractors. The Developer does not need to re-implement these routines.

## Extraction Routines

When an extractor is called to get some data, the DataGroup (discussed in INSTALLED PLUGIN IMPLEMENTATION REQUIREMENTS section below) is used to determine which call into the Base Extractor Class to make. These calls appear in CartExtractorBase as:

* protected virtual string **GetCatalog**(out int itemCount)

Used to get the retailer’s catalog of products. Both parent and child products (variants or options) are needed in the output. The output of this routine is saved to “upload/Catalog.txt”.

* protected virtual string **GetInventory**(out int itemCount)

Get the product catalog for those products that have been modified since the last extraction time.

Inventory vs catalog – Inventory refers to changes in product quantities and can be used to update the catalog to save time. Only the product id and inventory (stock count) fields of a product are saved to “upload/Inventory.txt”. In case of a large catalog that requires close inventory tracking, we pull the inventory on a more frequent schedule and only pull the entire catalog nightly or weekly.

* protected virtual string **GetSalesMonth**(DateTime exportDate, string filename, out int itemCount)

Used to get a single month of the retailer’s historical sales data (orders). The output of this routine is saved to “Sales-[Year-Month].txt”.

* protected virtual string **GetCustomers**(DateTime exportDate, string filename, out int itemCount)

Used to get and return the retailer’s Customer list. Output is saved to “upload/Customers\_Year\_Month.txt.

* protected virtual string **GetAtt1Names**(out int itemCount)

Used to get and return CategoryNames. Output is saved as “upload/Attribute1Names.txt”.

* protected virtual string **GetAtt2Names**(out int itemCount)

            Used to get and return BrandNames. Output is saved as “upload/Attribute2Names.txt”.

* protected virtual string **GetDepartmentNames**(out int itemCount)

Optional routine to get and return DepartmentNames. . Output is saved as “upload/DepartmentNames.txt”.

These calls can be found in CartExtractorBase.cs and are virtual so they can be overridden under extreme circumstances, but should not be overridden for most Direct API Implementations. Instead, lower-level methods and site rules should be used as described below.

The Extraction Routines use a set of helper classes called “**DataHandlers**”. There is one derived DataHandler class for each type of data being requested from the Platform API. The derived classes are:

**CatalogHandler** – used by GetCatalog()

**InventoryHandler** – used by GetInventory()

**SalesHandler** – used by GetSalesMonth()

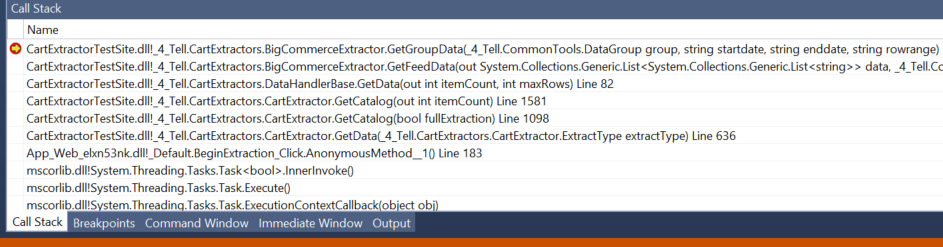
**CustomerHandler** – used by GetCustomers()

**AttributedHandler** – used by GetAtt1Names(), GetAtt2Names() and GetDepartments.

The DataHandlers are self-contained and platform agnostic and do not need to be modified by the Developer.

The Extraction Routines call the DataHandler.GetData() routine which in turn adds DataRange and RowRange specifications as refinements as needed and calls back to the CartExtractorBase class routine called GetFeedData() to create the URL and make the actual call to the client.

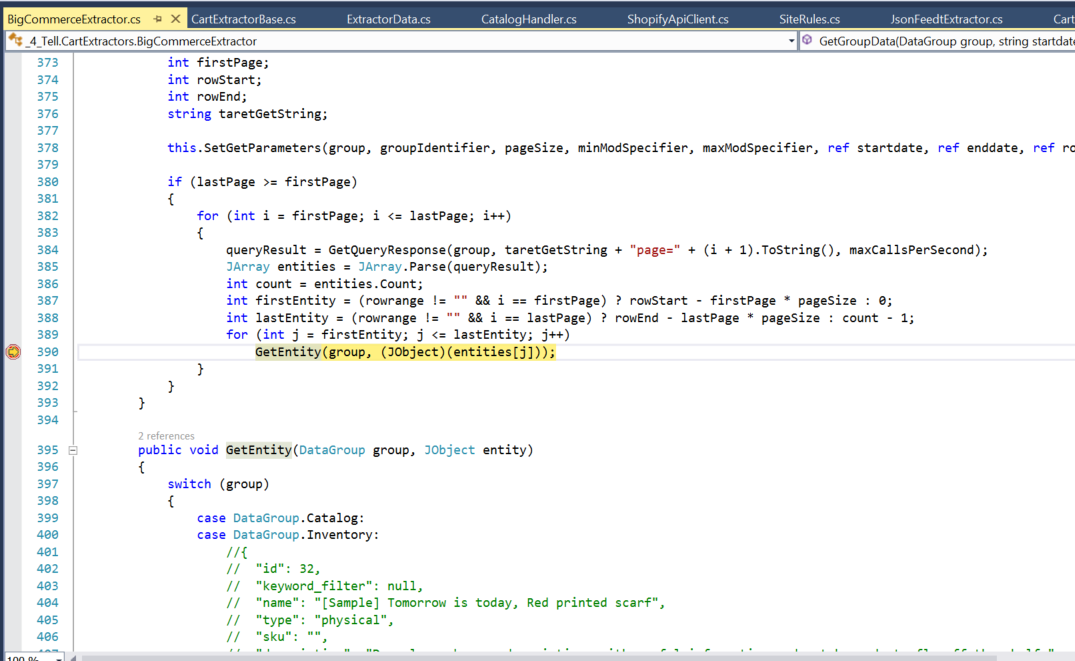
The Software Developer will usually need to override GetFeedData() in the new extractor. An example of overriding this routine can be found in the BigCommerce extractor. For example, when the call is made to get Catalog Data, the following call stack can be seen.



## Overriding GetFeedData()

The following override of GetFeedData() can be found in the example extractor you use as a template to build from. It is generic enough to use as is in your extractor.

It calls GetGroupData() as you can see in the call stack above. GetGroupData() can be used almost entirely as is in your extractor. The modifications that are required are covered in the “Build an Extractor – Step by Step” section below. The main function of GetGroupData() is to get the list of objects for a given group and walk that list processing each individual object. For instance it provides the loop to get the list of all orders and hands off processing of a particular order in the list. For generality we call a particular object an “entity” and call the routine that processes an entity, GetEntity(). You can see the loop here:



## GetEntity()

GetEntity() is where the bulk of the Developer’s work is done for a given extractor. Fortunately, the example that the Developer will use should be close in structure to what needs to be produced.

## Additional Base Class Overrides

In addition to the extraction routines the base class provides the following list of abstract functions that must be overridden in a specific extractor:

* public abstract bool ValidateCredentials(out string status);

Used to validate the API access credentials found in the site rules for the given retailer. The default override is:

public override bool ValidateCredentials(out string status)

{

throw new NotImplementedException();

}

* protected abstract void ReleaseCartData();

Used to free any large data structures built during the execution of an extractor. Not needed for all extractors. The Developer will need to override this routine to free any large objects that he creates in his derived extractor. The default override for a Direct API extractor is:

public override void ReleaseCartData()

{

}

* public abstract void LogSalesOrder(string orderID);

Only required if the platform’s order confirmation page does not include order details, which we use to create the auto actions file based on real time updates. Some platforms don't show order details on the confirmation page, they just provide an order index so our JavaScript can't scrape the order info we need. In this case our service comes here to find the order information based on the order id that was retrieved from the confirmation page. The default override is:

public override void LogSalesOrder(string orderID)

{

throw new NotImplementedException();

}

An example implementation is this one from the Bigcommerce extractor:

public override void LogSalesOrder(string orderID)

        {

            try

            {

                // Get the order

                queryResult = GetQueryResponse(DataGroup.Sales, string.Format("orders/{0}", orderID));

                if (queryResult == "")

                    throw new Exception(string.Format("Order id {0} not found", orderID));

                JObject order = JObject.Parse(queryResult);

                InitializeGroupData(DataGroup.Sales, "");

                GetEntity(DataGroup.Sales, order);

                if (sales.sales.Count == 0)

                    throw new Exception(string.Format("No products found for Order id {0}", orderID));

                foreach (Sale sale in sales.sales)

                {

                    DateTime oDate = (sale.Date.Length < 1) ?

Input.DateTimeConvert(DateTime.Now, Rules.SiteTimeZone) : DateTime.Parse(sale.Date);

                    int oQuantity;

                    if (!int.TryParse(sale.Quantity, out oQuantity) || oQuantity < 1) continue;

                    //Log in AutoActions-YY-MM.txt

    DataLogProxy.Instance.LogSingleAction(Alias, orderID, sale.ProductID, sale.CustomerID, oQuantity, oDate);

            }

        }

         catch (Exception ex)

          {

                if (Log != null)

                    Log.WriteEntry(EventLogEntryType.Error, "Error logging Sales Order", ex, Alias);

                return;

            }

        }

## Extraction Routine Output Format

To maintain generality between extractors, the data returned to the base class extractor when it calls the extraction routines must adhere to a specified format.

This format is very much like the format defined in the “Data Feed Response Format” section under INSTALLED PLUGIN IMPLEMENTATION REQUIREMENTS above. The data returned is, in fact, exactly the same, but there is one difference. Instead of an array of JSON string arrays ( where the first row of the array contains the column headers and the rest of the rows only hold the values) the data is reformatted into a list of lists of strings, one list of strings for each row of the JSON output, i.e. Extraction Routine Output = List<List<String>>.

Note that the requirements for output format are shared by both Extractor Implementations.

1. When the implementation is Direct API, the extractor routine output is the list of list of strings discussed in the paragraph above. That list of lists is passed to the extractor base class for final processing and writing to the txt files containing the tables expected by the generator.
2. When the implementation is Installed Plugin, the response output of the plugin must be as defined in the INSTALLED PLUGIN IMPLEMENTATION REQUIREMENTS section before it can be digested by the JsonFeedExtractor, but that extractor converts the JSON response into the same list of list of strings described above before passing to the extractor base class for final processing and writing.

## TXT File Output Format

Each of the extraction routines listed above will process the data it retrieves and output it as a table in a txt file under the “**upload**” directory of the retailer’s identifier. These files live on the server hosting the Dash Service under “**C:/ProgramData/4-Tell2.0/**”. For example if the retailer is <www.testsite.com> and his identifier is “**testsite**” then the upload directory used by the Extractor is “**C:/ProgramData/4-Tell2.0/testsite/upload**”.

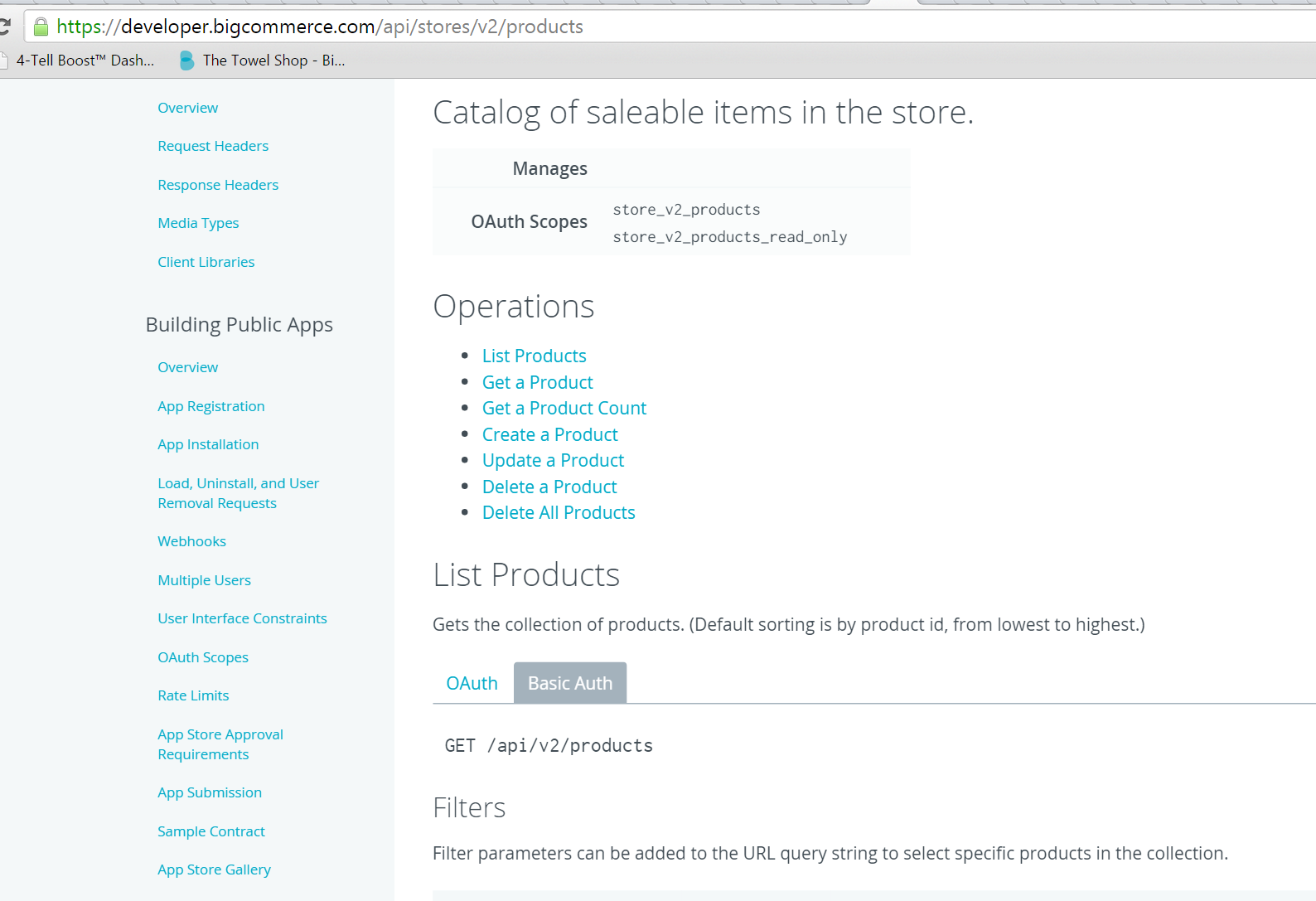
The table format and thus the output file format of each of the extractions routines are provided in:

**4-Tell Integration – Data Schema.pdf**

## Building a Direct API Extractor – Step by Step

This section walks the Developer through the steps he should take if he is using an existing Direct API extractor as an example template to build from:

1. Use GitHub to clone the master branch of 4-Tell/CartExtractorTestSite into your Visual Studio/projects directory.
2. Copy the Visual Studio/projects/CartExtractorTestSite/ProgramData directory to C:/ProgramData.
3. Locate the API documentation for the new target Platform. Study the API, determine what Authentication types and output formats it supports.
4. If it supports Basic Authentication and JSON output then use Bigcommerce as your example. Make a copy of CartExtractors/BigCommerceExtractor.cs, call it CartExtractor/YourPlatformExtractor.cs.
5. Bring up Visual Studio and open CartExtractorTestSite.sln.
6. Add CartExtrators/YourPlatformExtractor.cs to the CartExtractorTestSite project under the CartExtractors directory.
7. Edit YourPlatformExtractor.cs and replace all occurrences of “BigCommerce” with “YourPlatform”.
8. Choose a retailer that you want to test the new extractor against. Call it “TestSite.com” for this discussion. The Client Alias for the site will be “testsite”.
9. Create the directories C:/ProgramData/4-Tell2.0/testsite and C:/ProgramData/4-Tell2.0/testsite/upload.
10. Copy C:/ProgramData/4-Tell2.0/BigCommerceRules.xml to C:/ProgramData/4-Tell2.0/YourPlatformRules.xml.
11. Copy C:/ProgramData/4-Tell2.0/toweltuk /upload /SiteRules.xml to C:/ProgramData/4-Tell2.0/testsite/upload/SiteRules.xml. (toweltuk - [www.thetowelshop.co.uk](http://www.thetowelshop.co.uk) is an example Bigcommerce site)
12. Go back to the Platform’s API specification and identify the sections on Catalog (Products), Sales (Orders), Customers and CategoryNames (if available), BrandNames (if available). For example the Bigcommerce API section for Catalog looks like:



1. Determine the URLs used to access these data groups, e.g. <https://www.thetowelshop.co.uk/api/v2/products.json> for the Catalog. The form of the URL will vary by platform.
2. From the URLs determine the group name used to access data group. For example, Bigcommerce uses:

* “**products**” for Catalog
* “**orders**” for Sales
* “**customers**” for Customers
* “**categories**” for CategoryNames
* “**brands**” for Manufactures

1. Identify the access credentials for testsite. Edit C:/ProgramData/4-Tell2.0/testsite/upload/SiteRules.xml and update apiUrl, apiUser and apiKey to match the access credentials for testsite. For example:

  <apiUrl><https://www.TestSite.com/api/v2/></apiUrl>

  <apiUserName>4-tell</apiUserName>

  <apiKey>33e0daafe159b9c20d2ab42400b546b5</apiKey>

1. Go back to the API specification for each group and identify the names of the fields you want to extract for that group. For example, for Sales we want the order id, customer id of the customer ordering, date of the order, product id of the products ordered, and quantities of the products ordered. For example, in BigCommerce these are:

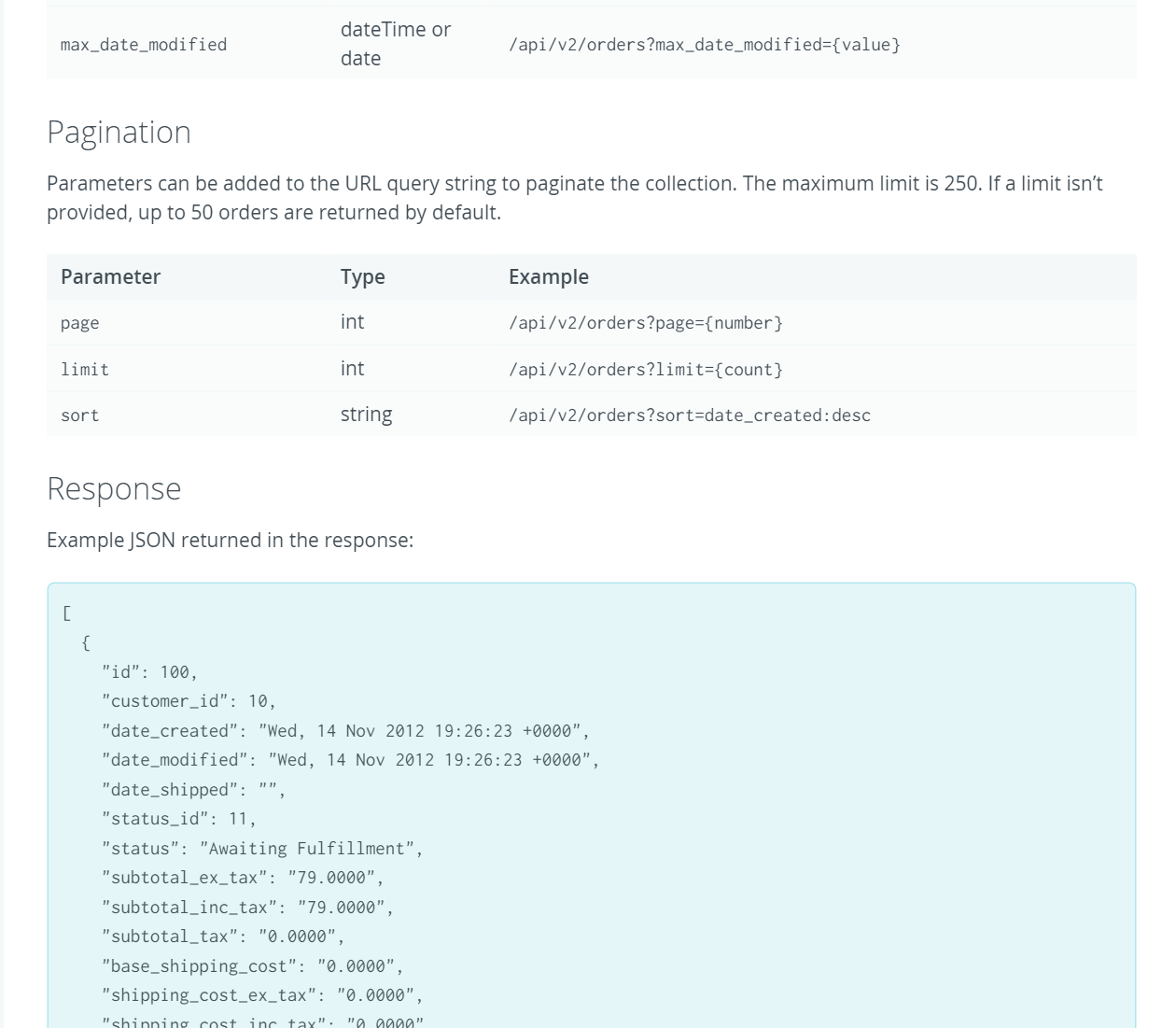
Order id – “id”

Customer id – “customer\_id”

Order Date – “date\_created”

Product id – “product\_id”

Product quantity – “quantity”



1. Enter these field names under the appropriate section in C:/ProgramData/4-Tell2.0/YourPlatformRules.xml. For example, for Sales we enter:

  <!-- orders (sales) -->

  <orderIdField>id</orderIdField>

  <orderProductIdField>product\_id</orderProductIdField>

  <orderCustomerIdField>customer\_id</orderCustomerIdField>

  <orderQuantityField>quantity</orderQuantityField>

  <orderDateField>date\_created</orderDateField>

These generic labels found in the rules (orderIdField, etc) will be used in the code to determine which fields to look for in the response from the API. E.g. consider the code in BigCommerceExtractor.cs - GetEntity() for processing the Sales entity (i.e. an order):

 try

 {

  string orderId = entity.SelectToken(Rules.Fields.GetName(FieldName.OrderId)).ToString();

  string customerId = entity.SelectToken(Rules.Fields.GetName(FieldName.OrderCustomerId)).ToString();

  DateTime temp = DateTime (entity.SelectToken(Rules.Fields.GetName(FieldName.OrderDate)).ToS

  string \_OrderProductIdName = Rules.Fields.GetName(FieldName.OrderProductId);

  string \_OrderQuantityName = Rules.Fields.GetName(FieldName.OrderQuantity);

  queryResult = GetQueryResponse(group, "orders/" + orderId + "/products.json", cond);

  JArray Products = JArray.Parse(queryResult);

  foreach (JObject product in Products)

  {

     Sale s = new Sale();

     s.ProductID = product.SelectToken(\_OrderProductIdName).ToString();

     s.Quantity = product.SelectToken(\_OrderQuantityName).ToString();

     s.OrderID = orderId;

     s.CustomerID = customerId;

     s.Date = temp.ToShortDateString();

     sales.AddSale(s);

  }

}

catch (RuntimeBinderException e)

You can see where we use Rules.Fields.GetName() to retrieve the values of the field names identified in the rules file. You can also see where we use SelectToken() with the retrieved values to get the data we want for the field in the JSON object returned from our query. Most of the entity processing in GetEntity() is very similar to this.

1. Replace the comments in GetEntity() with the relevant data structure for the target Platform. To do this go again to the Platform’s API specification and copy the structure shown for each entity into GetEntity() as a comment from which you can work while you code the processing for the entity.
2. Now modify each DataGroup case in GetEntity() as needed to gather the information from the entity. Note that several data structures are used to build the 4-Tell side copy of the entity. That copy is added to the list of objects (entities) returned from GetGroupData(). These data structures should not need to be modified by the Developer. They can be found in CartExtractors/ExtractorData.cs.
3. Modify the switch statement at the top of GetGroupData() according to the group names used to access each DataGroup. You will also update the page limit size for each group. This is the limit on the number of objects that can be requested from the API in any single call. Most platforms allow for up to 250 entities per request. Consult you Platform’s API spec for your limits.
4. Modify InitializeGroupData() based on what data needs to be gathered ahead of a group request before processing the response for that group can be completed. For instance, before processing catalog entities in BigCommerceExtractor.cs – GetEntity() we need to have custom fields and brands available, so we get them before we get the Catalog.
5. Go yet again to the Platform’s API specification and work with our Integration Specialist to determine whether we can scape Order Details from the Order Confirmation page of the site.
6. Locate LogSalesOrder in YourPlatformExtractor.cs and based on the answer to #22, either insert the default throw or implement the routine as seen in the example in BigCommerceExtractor.cs.
7. Test the extractor using the directions in the next section.

## Extractor Performance

There are several main factors affecting and controlling the performance of your extractor:

1. **Gating**: Most Platform APIs place restrictions on how fast you can access their interface. Usually this is quantified in the number of calls you can make per second. We keep a site rule called “maxCallsPerSecond” that defaults to 2 calls per second and can be overridden in the cartWebClientConfig section of your site’s SiteRules.xml.
2. **Request Limit**: Most Platform APIs place restrictions on how many records you can request from the API at one time. This number varies depending on the platform and the data group being requested. We set this limit in the switch statement at the beginning of GetGroupData(). There you will find a default setting called “pageSize” at the top of the switch, which can be overridden by setting it to another value on a case basis inside the switch. Consult your Platform API documentation to identify what your pageSize limits are.
3. **Number of API Calls**: Limiting the number of calls to the API can greatly reduce the overall time taken to get a data group. This can be accomplished by attempting to follow on simple rule – Try to structure your code so you never call GetQueryResponse() in GetEntity(). The reasoning behind this is as follows: GetEntity() is called for every object in the group of data you are gathering at the GetGroupData() level so making a call to GetQueryResponse() inside GetEntity() will mean you’re calling GetQueryResponse() on every object. This can be detrimental. To get around this attempt to identify the single call to the API that can gather all the data you need for the group you are requesting in GetGroupData() before you make that request. For instance, if customer address is not a field within the customer object (entity) and you therefore need to get the address for each customer, you could either make a call to the API when processing each customer in GetEntity() (you would supply the customerId when making that call) OR you could make a call (if it’s supported by your API) to get all customer addresses at once before you run GetGroupData() for the customers. That can be accomplished by adding a call to GetGroupData() for the customer addresses in the switch statement in InitializeGroupData(). E.g.:

public void InitializeGroupData(DataGroup group, string extraFields, string

        {

            FreeLocalStorage(false);

            switch (group)

            {

                case DataGroup.Catalog:

                    // Get customfields, categories and brands first to support catalog retrieval.

                    customfields = new CustomFields();

                    GetGroupData(DataGroup.Custom, "", "", "");

                    brands = new Brands(Rules);

                    GetGroupData(DataGroup.ManufacturerNames, "", "", "");

                    options = new Options();

                    GetGroupData(DataGroup.Options, "", "", "");

                    catalog = new Catalog(Rules, false);

                    catalog.SetHeader(extraFields.Split(',').ToList(), customfields.GetUniqueNames());

                    break;

                case DataGroup.Sales:

                    if (orderproducts == null)

                    {

                        orderproducts = new OrderProducts();

                        GetGroupData(DataGroup.OrderProducts, "", "", "");

                    }

                    sales = new Sales(Rules);

                    break;

                case DataGroup.Customers:

                    if (customeraddresses == null)

                    {

                        customeraddresses = new CustomerAddresses();

                        GetGroupData(DataGroup.CustomerAddresses, "", "", "");

                    }

                    customers = new Customers(Rules);

                    break;

## Testing an Extractor

A tool called the “**Cart Extractor Test Site**” should be used by the Developer to build and test his new extractor. It is a slimmed down version of the Dash Service that only runs extractors. The tool is a VS 2013 project that contains a directory called, “**CartExtractors/”**. In order to test a new extractor the following steps should be taken:

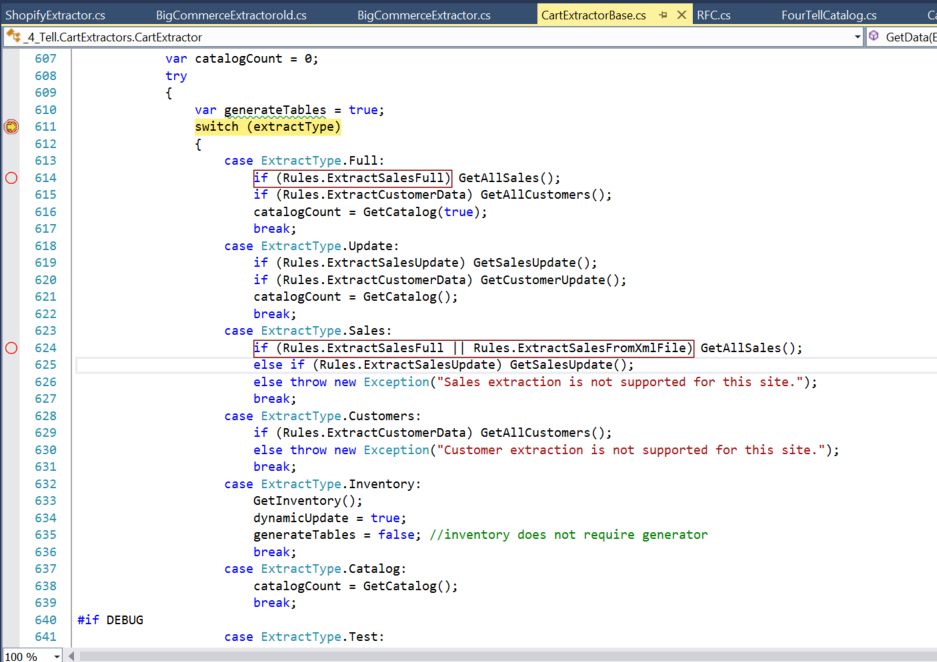
1. A new extractor should take the form **NewPlatformExtractor.cs** (replace “New Platform” with the name of the Platform) and be placed in the **CartExtractors** directory, added to the CartExtractorTestSite project and built into the Test Site application.
2. A Platform specific, default Site Rules file, **NewPlatformRules.xml** must be created locally under “**C:/ProgramData/4-Tell2.0**”.
3. To test the new extractor a test site using the target platform should be selected and its client alias identified. The developer should create a local directory in his dev environment called; “**C:/ProgramData/4-Tell2.0/new\_client\_alias/upload**”. One of these must be created for each alias used to test the new extractor against.
4. A new **SiteRules.xml** file must be created and placed in the “**upload**” directory for the site being tested. The CartExtractorBase class will attempt to read the site rules for the specified client alias from that location.

## Test Tool Operation

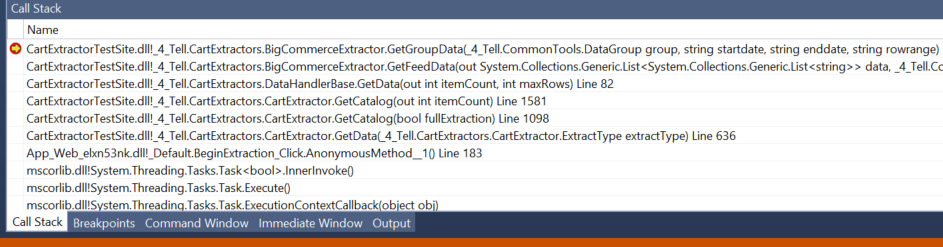
The tool provides a simple user interface containing the following UI controls:

1. Extraction Type dropdown allowing specification of the result type of extraction. This corresponds to the DataGroup specification under the Data Feed Definition section below. Possible selections are:
2. Full
3. Update
4. Catalog
5. Inventory
6. Sales
7. Customers

The result of each of these choices is guided by several other site rules according to the following code snippet from CartExtractorBase.GetData(). This represents the bottom of the call stack and originates the eventual calls to the derived extractor you are building.

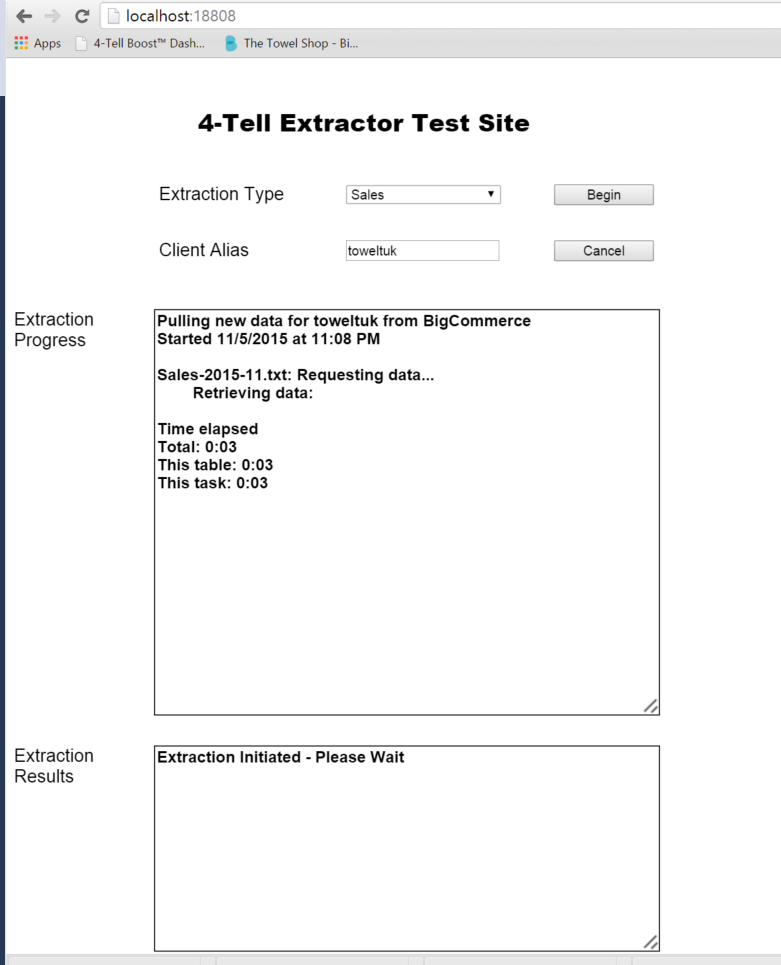


1. Client Alias entry field allowing specification of the shop to extract from. This corresponds to the ClientAlias specification under the Data Feed Definition section below. You will need to have the upload directory in place as described in the first paragraph of this section before you can access the SiteRules and write to the upload directory for this alias.
2. Begin Extraction button to start an extraction. This will make the call to GetData() in the base class as seen above and continue execution down into the derived class as shown in the call stack shown earlier:



What this shows is the eventual call into the derived extractor (BigCommerce in this case).

1. Cancel Extraction button to cancel an extraction.
2. Extraction Progress panel providing real-time updates of the extraction progress. There are several built-in facilities for logging and presenting the steps of the extraction process. These can be used by the Developer to present state information to the tool user.
3. Extraction Results panel providing overall status information.



**IMPORTANT NOTE:** A new extractor should be tested against the scenarios found under “Test Scenarios” in the INSTALLED PLUGIN IMPLEMENTATION REQUIREMENTS section.

## Supporting Installed Plugins

When a plugin is implemented as described previously in INSTALLED PLUGIN IMPLEMENTATION REQUIREMENTS, it provides a RESTful API that returns JSON according to “Data Feed Response Format” section found there. To consume that JSON output we include a “Direct API” extractor that reads in JSON (formatted as described) and exports the generator input tables as described above. This Extractor is called the JsonFeedExtractor and is used for all plugins created to support the Installed Plugin Implementation of the Extractor.

# JavaScript Insertion

JavaScript is inserted on all pages of the site in order to load our display code. The display code uses rules that the storeowner sets in their 4-Tell site plan to determine when and where to insert recommendations. Special JavaScript is also placed on the order confirmation page to send us order information. This information is used to track sales and enable transactional email recommendations.

## JavaScript for All Pages

The following JavaScript should be inserted into the main frame header so that it appears on all pages in the site. Note that the ClientAlias and Mode are variables, labeled CLIENTALIAS and MODE that must be made available from the Admin Settings above:

<!--4-Tell Recommendations Begin (www.4-tell.com)-->

<script type="text/javascript" async

src="//4tcdn.blob.core.windows.net/4tjs3/4TellLoader.js?alias=CLIENTALIAS&mode=MODE">

</script>

<!--4-Tell Recommendations End-->

If the site uses jQuery, and the version is below 1.7, we need to setup a NoConflict setting in the Javascript loader. The following code is used:

<!--4-Tell Recommendations Begin ([www.4-tell.com)--](http://www.4-tell.com)--)>

<script type="text/javascript"

src="//ajax.googleapis.com/ajax/libs/jquery/1.11.1/jquery.min.js">

</script>

<script type="text/javascript">

\_4TellBoost={}; \_4TellBoost.jq=jQuery.noConflict(true);

</script>

<script type="text/javascript" async

src="//4tcdn.blob.core.windows.net/4tjs3/4TellLoader.js?alias= SITE\_ALIAS">

</script>

<!--4-Tell Recommendations End-->

## Order Confirmation

Order confirmations allow the Boost service to track real-time sales behavior on the site. As such, the sales data feed is required to obtain the historical sales, but not required for the nightly update. Order confirmation also is a critical piece to allow recommendations in emails using our Boost4Email service.

Order Confirmations are accomplished by calling our UploadData/singleSale function on the order confirmation page and passing the details of the order that was just processed. **One call must be made for each item that was purchased in the order** since orders can have multiple items. This can either be done server-side or through JavaScript on client page. The call to our service should use the following format:

http://live.4-tell.net/Boost2.0/upload/singleSale?clientAlias={CLIENTALIAS}  
&orderID={ORDERID}&customerID={CUSTOMERID}&productID={PRODUCTID}  
&quantity={QUANTITY}

# Multi-Store & Multi-Contry

## Multiple Stores

If the platform allows several stores to share a catalog or run from the same admin panel, the feed is based on the Client Alias or API token as entered in the Admin information. This means that the admin panel must allow separate entries for the ClientAlias or API token for each store or site. The storeowner can then decide whether to keep the same alias for all stores or purchase separate aliases for each store so that the catalog and behavior tracking will be handled separately. If a Client Alias is entered on the master store, the dependent stores use that Client Alias, unless the retailer changes the settings for the dependent stores. As such, the Client Alias can be all different, all the same, or any combination for multiple stores.

When the Boost Service calls the Feed URL for data, it provides a ClientAlias or API token as part of the request. If more than one store is setup with a particular alias, then the data for all stores with that alias should be returned.

If the retailer has disabled a store (either in a 4-Tell setting or for the platform), even if it has a valid Client Alias, requests for recommendations should NOT be made from that store. The feed, however, should still include data from that store.

The Client Alias can be blank, and a store with a blank Client Alias should neither (i) have data included in any feed nor (ii) request recommendations.

The Service Key may be different for each Client Alias in multi-store, but preferably the same.

## Multiple Countries (FUTURE)

If the same store is designed for multiple countries, where the language and currency (and possible images) change for the country, we pull the catalog feed for each country. We only pull the changed elements in the catalog. For example, if only the currency and language change, we only pull the price, sale price, list price, cost and name.

The number of countries is included in the Version response (see above). This is used by 4-Tell to request the catalog for each country using the CountryIndex (see above). It is assumed that the platform has a corresponding index that is used for each country.

This index is critical because, in the front end, it must be captured in JavaScript and passed in the call to our service to determine the correct catalog for the response.

# Contact Information

### Sales:

[sales@4-tell.com](mailto:sales@4-tell.com) or (503) 746-9070 x 2

### Technical Support:

[support@4-Tell.com](mailto:support@4-Tell.com) or (503) 746-9070 x 3