Saturday, August 30, 2014

Loading Orders with AMTU

Overview

This document provides a simple overview of using Amazon's AMTU utility to download Order data and load the data into SQL Server on Windows Server 2012. (You can easily replace SQL Server with mySQL, but the specific code examples presented in this version use SQL Server T-SQL syntax). You could also do this with a shell script on Mac OS or Linux. This is "a" way of getting this data, not necessarily the "best" way.

Document History

Version	Date	Description
1.0	8/30/2014	Initial Version

Amazon Requirements

This document assumes that you already have an Amazon MWS account. This is more than just an account on seller central, as it allows you to use "web services" with Amazon. If you don't have an MWS account then use the following link to sign up for MWS:

https://sellercentral.amazon.com/gp/mws/registration/register.html

Once you have registered for an Amazon MWS account you will need the AMTU software. AMTU is an Amazon-supplied software program (Windows, Linux and Mac OS) that automatically uploads and downloads data files. Get Amazon's AMTU at the following link:

http://www.amazon.com/gp/help/customer/display.html?nodeId=200779340

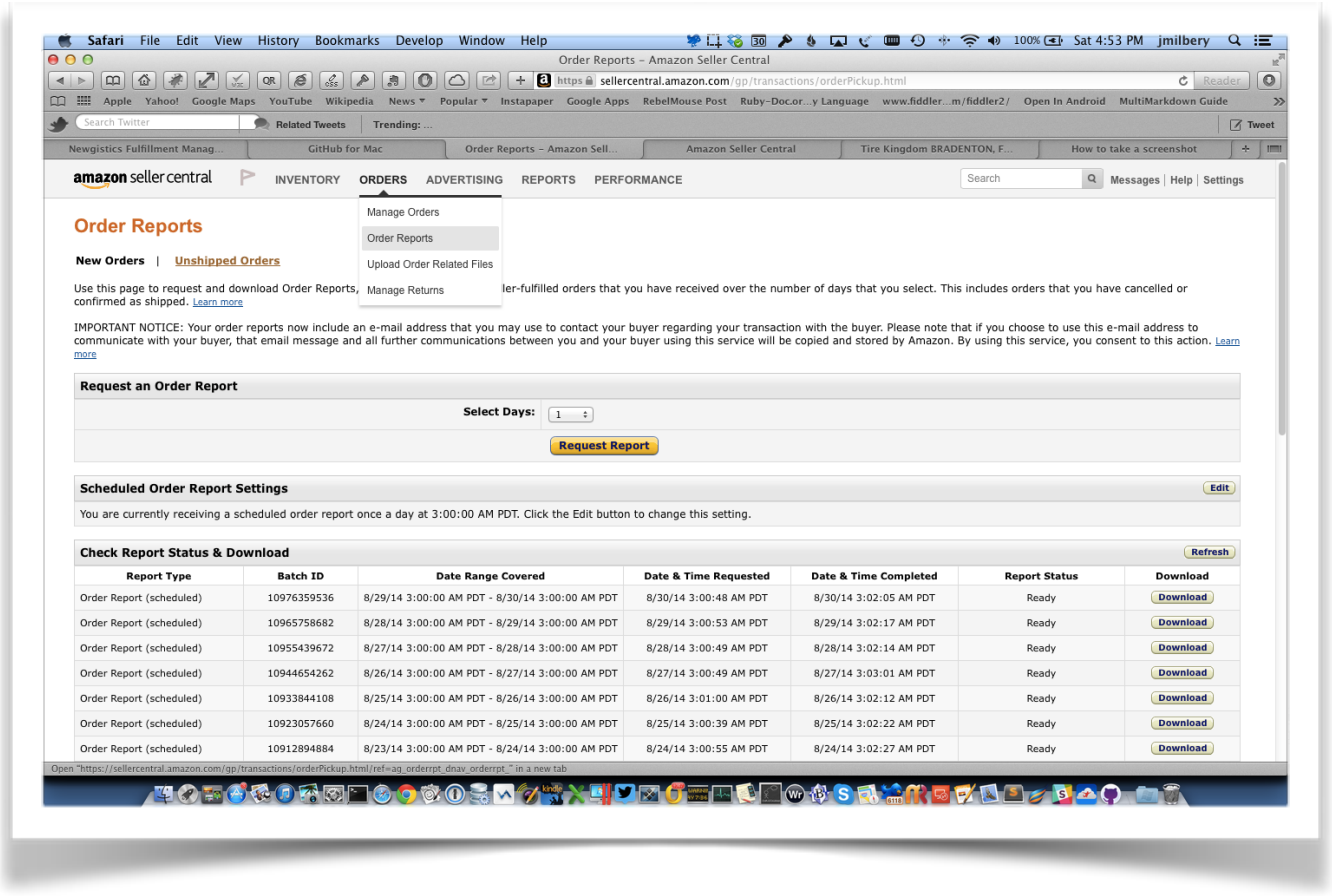
This guide doe not walk you through installing and configuring AMTU — there is a great user guide for that.

The rest of this document assumes that you've got AMTU up and running and that you have your MWS account.

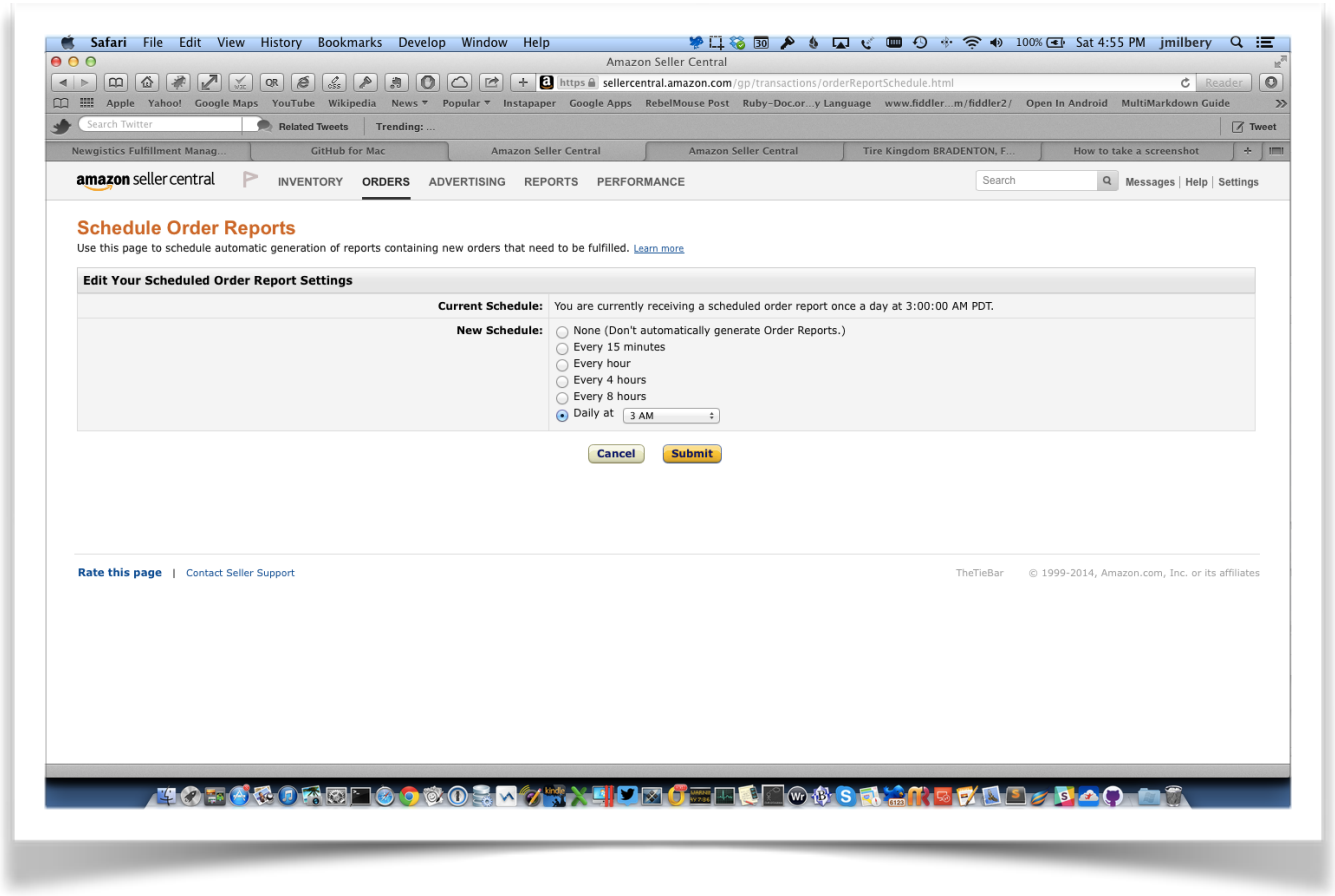
Configuring Amazon via Seller Central

Once you have AMTU up and running, you will need to tell Amazon to deliver order files to you on a schedule — this can be done through seller central.

- Logon to seller central and choose the "ORDERS" menu and then the "Order Reports" Option as shown below



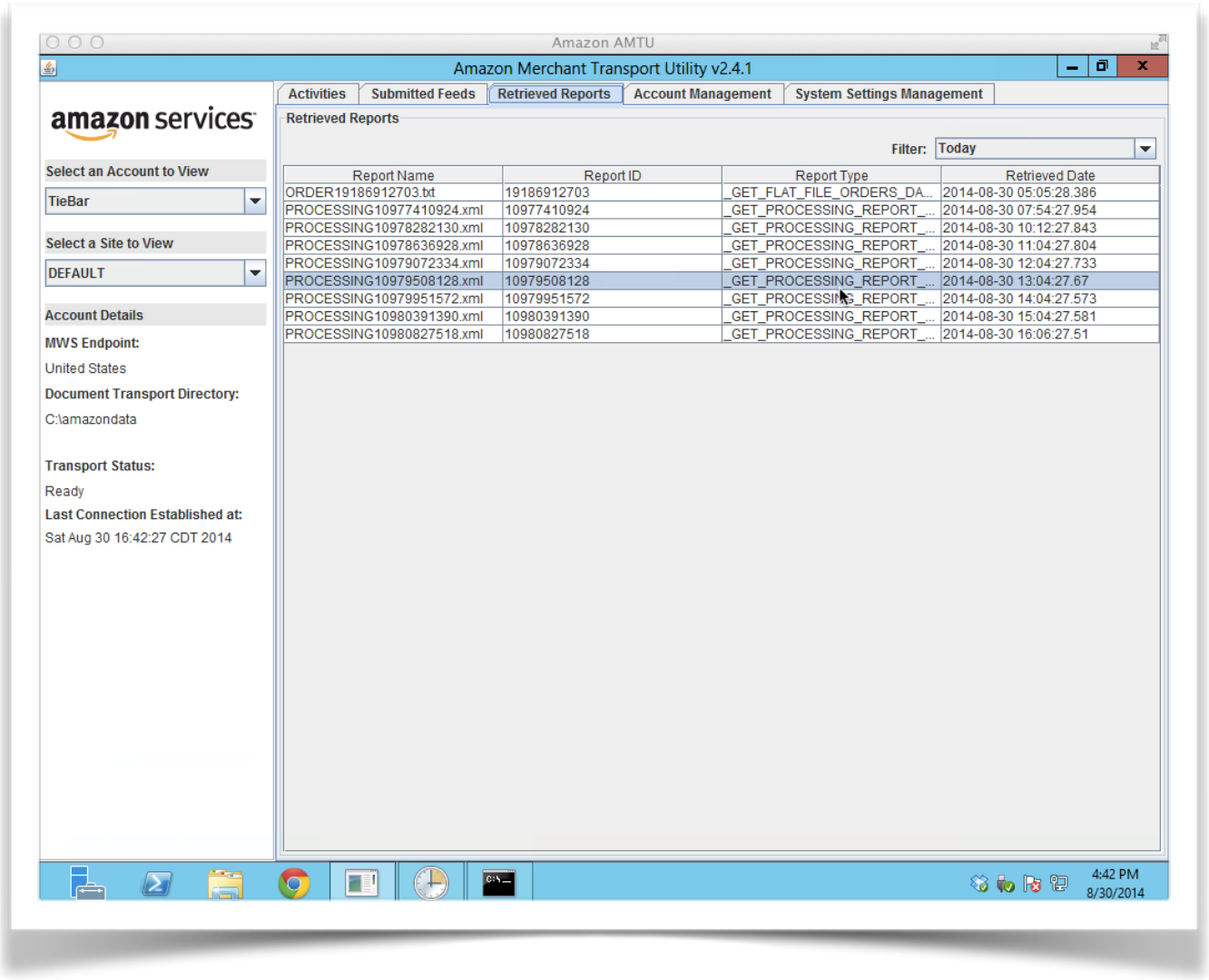
- Choose the “edit” button (or “new”) on the far right (also shown in the preceding panel in the middle on the far right). This will bring up the menu as shown below:



- Select a choice from the “New Schedule” option. In this case, we’ve chosen to have Amazon send us orders every day at 3 am.

AMTU

Once you've configured Amazon to deliver order reports on a schedule AND installed AMTU, you are ready go.



AMTU will automatically "wake up" and download files on a schedule. By default it checks for new files every 5 minutes. This is overkill in our case, since we are only exporting ORDERS files at 3 am — once a day. However, you can also use AMTU to upload inventory status, order confirmation and shipping confirmation files - so it makes sense for AMTU to "check in" on a more frequent schedule.

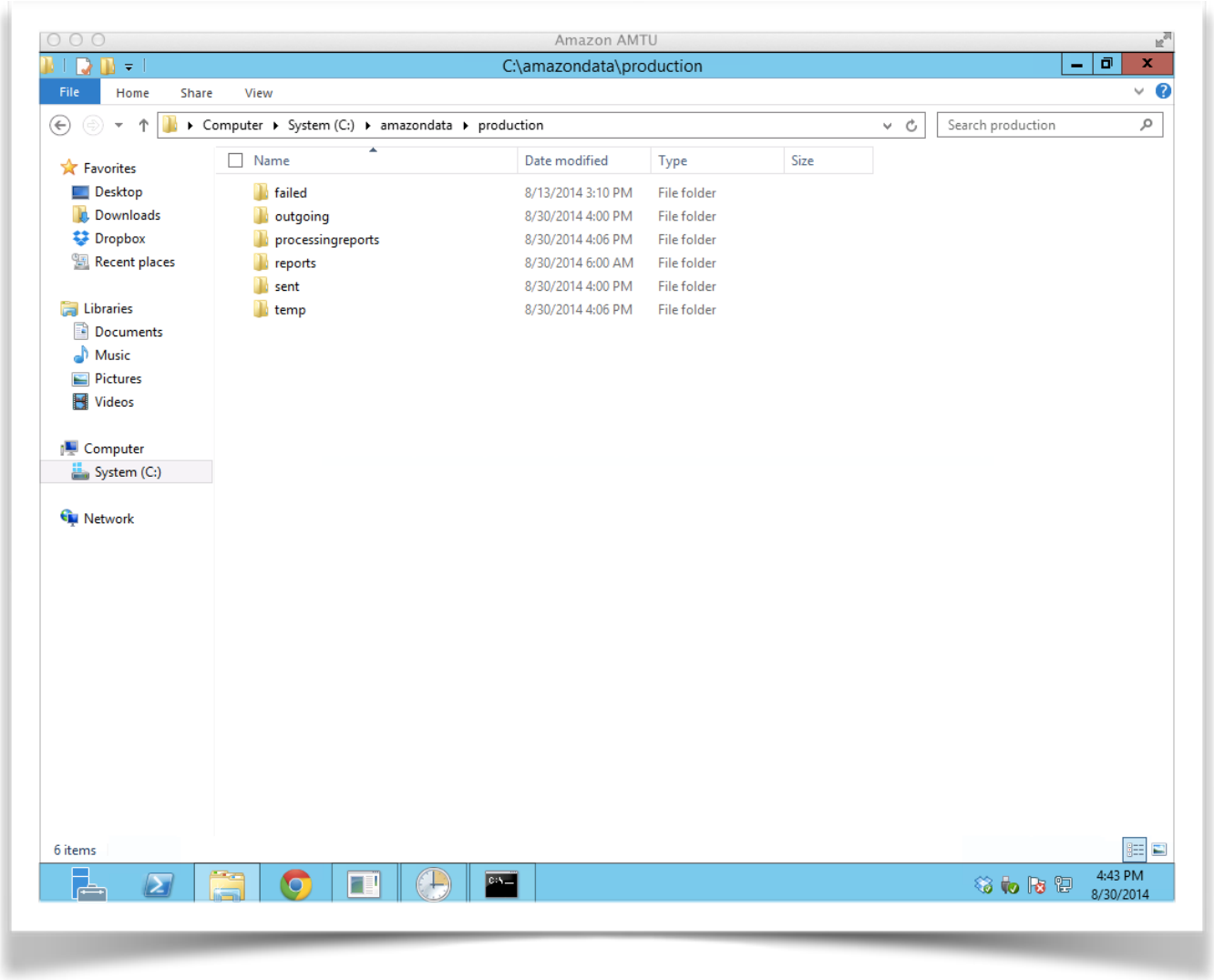
AMTU has three main tabs that you will work with.

Activities — Activities is a log of all of AMTU's activities. This panel gets updated whenever AMTU does anything, when it downloads files or you upload files, etc.

Submitted Feeds — We don't use this panel when we download orders, but we will use it for subsequent code snippets. This panel gets updated when you submit files for processing.

Retrieved Reports — When AMTU fetches files for you it updates this panel. As you can see in the preceding screenshot, AMTU has "fetched" a number of reports today — a number of processing reports and a single "Orders" report - at 3 am when we requested it. (The log shows 5 am in the screen shot because AMTU is running on Seattle time and our installation of AMTU is running at Rackspace on Chicago time - 3 am in Seattle = 5 am in Chicago).

The key to understanding AMTU is to understand AMTU's directory structure.



We've configured AMTU to use the directory "c:\amazondata" — which causes AMTU to create a series of directories as shown in the figure above. AMTU stores all files that it downloads (like ORDERS) in the **production\reports** subdirectory.

So, when AMTU wakes up and checks your account at 3 am (in our example) — it will download your ORDERS file into the **production\reports** directory.

The files will not be called “ORDERS.TXT” — rather, they will have a long name that includes a unique timestamp such as “**ORDER19186912703.txt**”. And this is exactly where the fun begins.

Amazon ORDER###### Files

AMTU downloads your ORDERS files as TAB-delimited TEXT files using the naming convention “ORDER##########.TXT”.

It’s up to you to figure out the name and process the file.

First, let’s start with the file itself. In this repository there is a file called “ORDERS.TXT” — it is a snippet of an Amazon orders file as follows:

```
order-id|order-item-id|purchase-date|payments-date|buyer-email|buyer-
name|buyer-phone-number|sku|product-name|quantity-purchased|currency|
item-price|item-tax|shipping-price|shipping-tax|ship-service-level|
recipient-name|ship-address-1|ship-address-2|ship-address-3|ship-city|
ship-state|ship-postal-code|ship-country|ship-phone-number|tax-
location-code|tax-location-city|tax-location-county|tax-location-
state|per-unit-item-taxable-district|per-unit-item-taxable-city|per-
unit-item-taxable-county|per-unit-item-taxable-state|per-unit-item-
non-taxable-district|per-unit-item-non-taxable-city|per-unit-item-non-
taxable-county|per-unit-item-non-taxable-state|per-unit-item-zero-
rated-district|per-unit-item-zero-rated-city|per-unit-item-zero-rated-
county|per-unit-item-zero-rated-state|per-unit-item-tax-collected-
district|per-unit-item-tax-collected-city|per-unit-item-tax-collected-
county|per-unit-item-tax-collected-state|per-unit-shipping-taxable-
district|per-unit-shipping-taxable-city|per-unit-shipping-taxable-
county|per-unit-shipping-taxable-state|per-unit-shipping-non-taxable-
district|per-unit-shipping-non-taxable-city|per-unit-shipping-non-
taxable-county|per-unit-shipping-non-taxable-state|per-unit-shipping-
zero-rated-district|per-unit-shipping-zero-rated-city|per-unit-
shipping-zero-rated-county|per-unit-shipping-zero-rated-state|per-
unit-shipping-tax-collected-district| per-unit-shipping-tax-collected-
city|per-unit-shipping-tax-collected-county|per-unit-shipping-tax-
collected-state|item-promotion-discount|item-promotion-id|ship-
promotion-discount|ship-promotion-id|delivery-start-date|delivery-end-
date|delivery-time-zone|delivery-Instructions|sales-channel
104-11111111-1111111|222222222222222|2014-08-29T04:54:01-07:00|
2014-08-29T04:54:01-07:00|xxxxxxxxx@marketplace.amazon.com|Joe
Customer|2222222222|SKU123456|SKU123456 Description|2|USD|17.98|0.00|
5.99|0.00|Standard|Joe Customer|123 Main Street|||Chicago|IL|
```

```
60606-1111|US|1234567890|||||||||||||||||||||||||||||||||||||||0.00||
0.00||||||
```

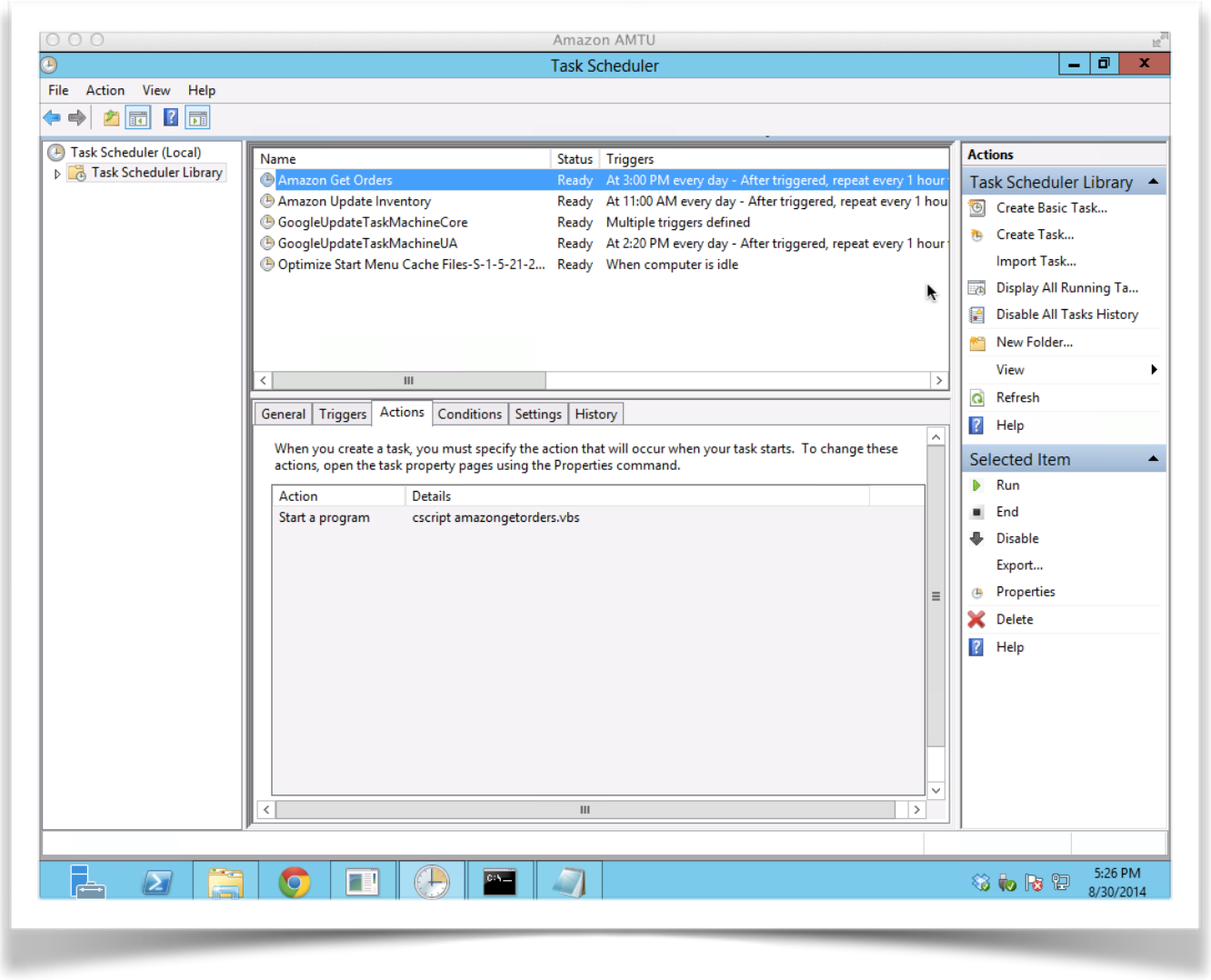
Our sample orders file has been modified SLIGHTLY from the standard Amazon orders file format. First, we've replaced the "tab" characters with vertical bars ("|") so that you can see the data more easily in this document. In the repository that comes with this document we kept the tab characters as they were. Second, we changed the data to protect the innocent. Third, we deleted everything but a single record in order to make it easier to display and discuss the data.

In most cases you will not need all of the columns in the report, but we load them all just the same.

The most important thing to note is that the ORDERS files have duplicate data. If a person orders more than one item from you — you will see the same order number with different order item data - but the shipping information will repeat. In common parlance, the data is in spreadsheet format not relational database format. So you will HAVE to post-process the data after you get it into the database.

Fetching the Data

AMTU will automatically download the ORDERS#####.TXT files for you. All that you need to do is to “wake up” and process the files.



For this example we used the Windows Task Scheduler — but you can just as easily use CRON on Linux or Mac OS.

We have our Amazon job set to run at 3 PM (just for the purposes of example — it does not have to match the AMTU schedule).

When it executes, it runs the “**amazongetorders.vbs**” — and this is part of the secret sauce.

We are using VBScript, which is built into Windows Server — but you could just as easily write a shell script on Linux and/or Mac OS.

```
Set fso=CreateObject("Scripting.FileSystemObject")
Set objShell = WScript.CreateObject("WScript.Shell")
dim bcpname
Set fldr=fso.getFolder("c:\amazondata\production\reports")
for each file in fldr.files
if instr(file,"ORDER") >0 then
     bcpname = "bcp amazondb.dbo.amazonORDERSTXT in " & file & "
-c -F 2 -S 10.123.3.100 -m 1 -U admin -P abc123"
     wscript.echo bcpname
     Return = objShell.Run(bcpname, 1, true)
     fso.MoveFile file, "C:\amazoncomplete\"
end if
next
```

The Windows task that we run everyday at 3pm is “cscript amazonGetOrders.vbs” — the text of this script is shown above.

Windows provides “wscript” and “cscript” as VBscript processors - the first provides a windows interface while the latter provides a command-line interface. We are using “cscript” — the command line — since we are running as a background task without any user interface.

Our VBScript is inelegant and simple. I’ve highlighted the lines that you will need to change in bold typeface. Otherwise, the script will run as shown (and as provided in this github repo).

The first three lines instantiate variables for the script. The fourth line instructs VBScript to look in a directory. You will need to modify the string to point to YOUR amazon production directory.

The script loops through every file in that directory. For each file, if the name has the string “ORDER” in the name, then we know that it is an AMAZON orders file.

If the file is an ORDER file, we load it into SQL Server using “bcp” — SQL Server’s bulk copy program. We’ll talk about the table structure and BCP in the next section. For now, just understand that our script creates a string that includes a number of BCP instructions.

It runs these instructions using the “objShell.Run” syntax and then moves the file that we’ve processed to another directory — so we won’t process it again.

BCP and the SQL Server table

Our script relies on a table structure in SQL Server as follows:

```
CREATE TABLE [dbo].[amazonORDERSTXT](
      [order_id] [varchar](50) NULL,
      [order_item_id] [varchar](150) NULL,
      [purchase_date] [varchar](25) NULL,
      [payments_date] [varchar](50) NULL,
      [buyer_email] [varchar](150) NULL,
      [buyer_name] [varchar](150) NULL,
      [buyer_phone_number] [varchar](20) NULL,
      [sku] [varchar](512) NULL,
      [product_name] [varchar](100) NULL,
      [quantity_purchased] [varchar](5) NULL,
      [currency] [varchar](10) NULL,
      [item_price] [varchar](10) NULL,
      [item_tax] [varchar](10) NULL,
      [shipping_price] [varchar](10) NULL,
      [shipping_tax] [varchar](10) NULL,
      [ship_service_level] [varchar](25) NULL,
      [recipient_name] [varchar](200) NULL,
      [ship_address_1] [varchar](100) NULL,
      [ship_address_2] [varchar](100) NULL,
      [ship_address_3] [varchar](100) NULL,
      [ship_city] [varchar](100) NULL,
      [ship_state] [varchar](100) NULL,
      [ship_postal_code] [varchar](100) NULL,
      [ship_country] [varchar](10) NULL,
      [ship_phone_number] [varchar](25) NULL,
      [tax_location_code] [varchar](10) NULL,
      [tax_location_city] [varchar](10) NULL,
      [tax_location_county] [varchar](10) NULL,
      [tax_location_state] [varchar](10) NULL,
      [per_unit_item_taxable_district] [varchar](10) NULL,
      [per_unit_item_taxable_city] [varchar](10) NULL,
      [per_unit_item_taxable_county] [varchar](10) NULL,
      [per_unit_item_taxable_state] [varchar](10) NULL,
      [per_unit_item_non_taxable_district] [varchar](10) NULL,
      [per_unit_item_non_taxable_city] [varchar](10) NULL,
      [per_unit_item_non_taxable_county] [varchar](10) NULL,
      [per_unit_item_non_taxable_state] [varchar](10) NULL,
      [per_unit_item_zero_rated_district] [varchar](10) NULL,
      [per_unit_item_zero_rated_city] [varchar](10) NULL,
      [per_unit_item_zero_rated_county] [varchar](10) NULL,
      [per_unit_item_zero_rated_state] [varchar](10) NULL,
      [per_unit_item_tax_collected_district] [varchar](10) NULL,
      [per_unit_item_tax_collected_city] [varchar](10) NULL,
      [per_unit_item_tax_collected_county] [varchar](10) NULL,
      [per_unit_item_tax_collected_state] [varchar](10) NULL,
```

```
[per_unit_shipping_taxable_district] [varchar](10) NULL,
[per_unit_shipping_taxable_city] [varchar](10) NULL,
[per_unit_shipping_taxable_county] [varchar](10) NULL,
[per_unit_shipping_taxable_state] [varchar](10) NULL,
[per_unit_shipping_non_taxable_district] [varchar](10) NULL,
[per_unit_shipping_non_taxable_city] [varchar](10) NULL,
[per_unit_shipping_non_taxable_county] [varchar](10) NULL,
[per_unit_shipping_non_taxable_state] [varchar](10) NULL,
[per_unit_shipping_zero_rated_district] [varchar](10) NULL,
[per_unit_shipping_zero_rated_city] [varchar](10) NULL,
[per_unit_shipping_zero_rated_county] [varchar](10) NULL,
[per_unit_shipping_zero_rated_state] [varchar](10) NULL,
[per_unit_shipping_tax_collected_district] [varchar](10) NULL,
[per_unit_shipping_tax_collected_city] [varchar](10) NULL,
[per_unit_shipping_tax_collected_county] [varchar](10) NULL,
[per_unit_shipping_tax_collected_state] [varchar](10) NULL,
[item_promotion_discount] [varchar](10) NULL,
[item_promotion_id] [varchar](10) NULL,
[ship_promotion_discount] [varchar](10) NULL,
[ship_promotion_id] [varchar](10) NULL,
[delivery_start_date] [varchar](10) NULL,
[delivery_end_date] [varchar](10) NULL,
[delivery_time_zone] [varchar](10) NULL,
[delivery_Instructions] [varchar](10) NULL,
[sales_channel] [varchar](10) NULL)
```

This script is stored in the githib repo as “table_amazonORDERSTXT.sql”.

The database that we have defined stores everything as a VARCHAR string - this makes it easy to bulk copy into the database. But, this also means that you will need to write some post-processing routines in order to extract the data from this “text” table and load it into your production database tables.

We define a string of BCP commands that the script executes. In order for this to run you will have to make sure that your machine has the SQL Server client tools installed.

Let’s walk through the various components of the BCP command:

bcp - The name of the BCP utility (bcp.exe)

amazondb.dbo.amazonORDERSTXT - This is the database, owner and tablename that we are going to copy the data INTO

in - denotes that we are copying “in” as opposed to “out”

“ & file & " - This is part of the VBScript, it will be replaced with the NAME of the file to load

—c - denotes that everything will be a character field

—F 2 - denotes that we will start loading the file at the 2nd row (in order to skip the column names in the file)

—S 10.123.3.100 - Server name (in our case an IP address)

—m 1 — Number of errors before the script terminates, we are using "1" as it should not error out if we load as character data

-U admin - Username

—P abc123 — Password of user.

* **You can replace the -U and -P parameters and their data with "-T" if you want to connect to SQL Server with a Trusted Windows Account.**

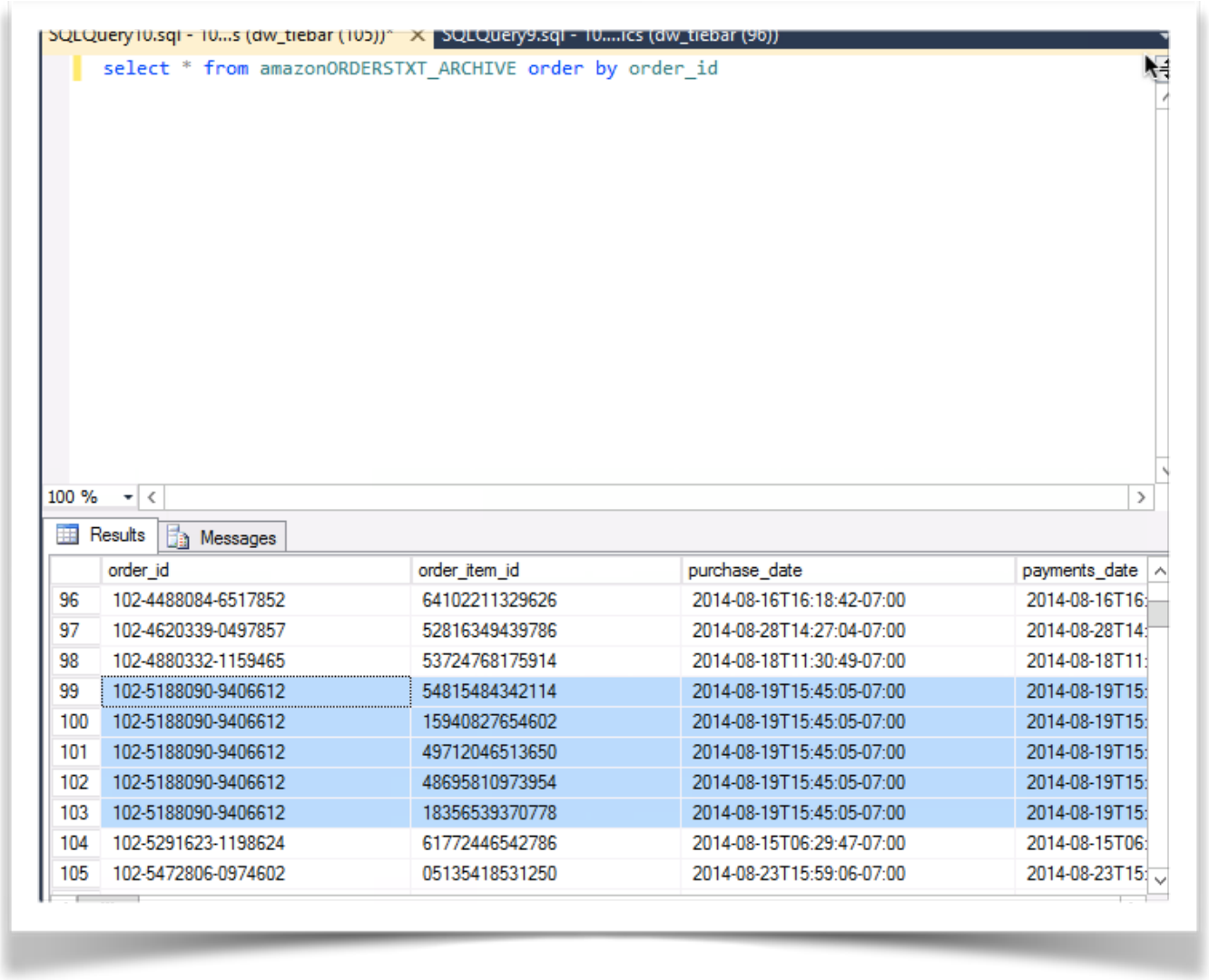
For our working example this command would look like the following:

```
bcp amazondb.dbo.amazonORDERSTXT in ORDER19186912703.txt -c -F 2
-S 10.123.3.100 -m 1 -U admin -P abc123"
```

BCP will execute and copy the data into the amazondb.dbo.amazonORDERSTXT table in the database.

Post Processing

BCP copies the data into SQL Server as strings — using our script and table format. This is the simplest way to get the data into SQL Server — but probably NOT the format that you need for your internal ERP or order-entry system.



	order_id	order_item_id	purchase_date	payments_date
96	102-4488084-6517852	64102211329626	2014-08-16T16:18:42-07:00	2014-08-16T16:
97	102-4620339-0497857	52816349439786	2014-08-28T14:27:04-07:00	2014-08-28T14:
98	102-4880332-1159465	53724768175914	2014-08-18T11:30:49-07:00	2014-08-18T11:
99	102-5188090-9406612	54815484342114	2014-08-19T15:45:05-07:00	2014-08-19T15:
100	102-5188090-9406612	15940827654602	2014-08-19T15:45:05-07:00	2014-08-19T15:
101	102-5188090-9406612	49712046513650	2014-08-19T15:45:05-07:00	2014-08-19T15:
102	102-5188090-9406612	48695810973954	2014-08-19T15:45:05-07:00	2014-08-19T15:
103	102-5188090-9406612	18356539370778	2014-08-19T15:45:05-07:00	2014-08-19T15:
104	102-5291623-1198624	61772446542786	2014-08-15T06:29:47-07:00	2014-08-15T06:
105	102-5472806-0974602	05135418531250	2014-08-23T15:59:06-07:00	2014-08-23T15:

You will note above that we are DUPLICATING header data — you can see the same ORDER_ID repeated for EACH item ordered.

In our sample case we have a SQL Server agent job that processes the ORDERS string data into the proper format and loads it into our ERP tables.

For the purposes of example we have SIGNIFICANTLY simplified the script (removing error handling code and anything that makes the script harder to ready.

I am providing this script simply as a VERY simple way of moving the data from the flat string format into production database tables and not as the optimal way to do so.

```
/*
** ng_move_amazon_orders
** Auth: JFM
** Desc: Stored procedure to pull amazon orders into the orders
** table for processing
** Usage: exec ng_move_amazon_orders
**
** Revision History
**
**    V1.0   08/15/2014 -- Created
**    V1.1   08/19/2014
**
*/
create procedure [dbo].[ng_move_amazon_orders] @showVersion int =
0
as
begin
declare @v_throw_error varchar(2000);
if @showVersion = 1
begin
     select @v_throw_error = 'NG_MOVE_AMAZON_ORDERS V1.1
8/19/2014';
     THROW 60001,@v_throw_error,1
end
set nocount on;

/*
** Variables for start of <ORDER> segment and all of
<CUSTOMERINFO> segment
*/
declare @v_orderid int;
declare @v_amazonOrderID varchar(50);
    declare @v_number_of_amazonorderstxt int;

/*
** Fix data in the string holding table prior to moving
** it into the production tables.  Use this to modify
** data in the import table PRIOR to moving the data into
** your orders/items tables.
*/
update newgistics.dbo.amazonorderstxt set ship_service_level =
upper(replace(ship_service_level,' ',''));
update newgistics.dbo.amazonorderstxt set ship_service_level =
'NGSPS' where ship_service_level='STANDARD'
and ship_country = 'US';
```

```
update newgistics.dbo.amazonorderstxt set ship_service_level =
'UPS2D' where ship_service_level='2DAY'
and ship_country = 'US';
update newgistics.dbo.amazonorderstxt set ship_service_level =
'UPS1S' where ship_service_level='1DAY'
and ship_country = 'US';
update newgistics.dbo.amazonorderstxt set ship_service_level =
'MSIIP' where ship_service_level='STANDARD'
and ship_country = 'CA';
update newgistics.dbo.amazonorderstxt set ship_service_level =
'UPSWS' where ship_service_level='EXPEDITED'
and ship_country = 'CA';
update newgistics.dbo.amazonorderstxt set ship_service_level =
'UPSWS' where
ship_country not in ('US','CA');


/*
** You could/should use cursors for this portion, but
** we are cheating a bit.  We just get a count of records
** to process and then we loop through them.  We add all the
** the order headers, get the order id numbers generated
** by the insert, join them to the items data from the
** imported string data in a temp table and then push
** them into the orderitems table.
*/
select @v_number_of_amazonorderstxt =
     isnull(count(distinct order_id),0) from
     amazondb.dbo.amazonorderstxt;
while @v_number_of_amazonorderstxt >0
begin
     /*
     ** New records -- load them into the temp orders table
     */
     select top 1 @v_amazonOrderID =
     left(order_id,50) from newgistics.dbo.amazonorderstxt
     where order_id not in
          (select amazonOrderid from
               thetiebar.dbo.orders where
                    amazonorderid is not null);

     insert into production.dbo.orders
     (amazonOrderID, OrderType, custId, orderDate, secureId,
     shippingType, shipping, tax, shippingAddressType,
     shippingFirstName, shippingLastName, shippingAddress,
     shippingAddress2, shippingCity, shippingState,
     shippingZipCode, shippingCountry, shippingPhone, email,
     statusId, NgShipmentId, approved)
          select top 1 order_id, 'amazonfbttb', order_id as
               custid, cast(left(purchase_date,10) as date),
               order_id as secureId, ship_service_level,
               cast(shipping_price as money),
               0.00,'Residential',' ',
```

```
                  left(recipient_name, 100),
                  left(ship_address_1,100),
                  left(ship_address_2,100),
                  left(ship_city,100), left(ship_state,50),
                  left(ship_postal_code,50),left(ship_country,50),
                  left(ship_phone_number,50),
                  left(buyer_email,100), 1, 20, 0
                  from
                        amazondb.dbo.amazonorderstxt where
                        order_id = @v_amazonOrderID;

      set @v_number_of_amazonorderstxt =
            @v_number_of_amazonorderstxt - 1
      end


/*
** Orders in place, add orderitems in bulk.
*/
select o.id as orderid, a.sku, cast('' as varchar(100)) as
      productnumber, a.quantity_purchased as quantity,
      a.product_name as productdescription,
      a.item_price as price, o.secureid, 0 as term,0 as periodid
into #tmp_amazonorderitems
      from production.dbo.orders o join
      amazondb.dbo.amazonORDERSTXT a
      on o.amazonorderid = a.order_id
      and o.id not in (select orderid from
            production.dbo.orderitems);
/*
** Replace Product Numbers (Our internal system has a different
** sku than the sku that we use for Amazon — this table maps
** skus from amazon to our internal skus.
*/
update o
set productnumber = pv.sku
from #tmp_amazonorderitems o left join
production.dbo.productvariant pv on o.sku = pv.amazonsku;
/*
** Load into the items table
*/
insert into production.dbo.orderitems
(orderid, productnumber, quantity, productDescription,
price, secureId, term, periodid)
select orderid, isnull(productnumber,'AMAZONUNKNOWN'), quantity,
productDescription,
price, secureId, term, periodid
from #tmp_amazonorderitems
end
```

d