

# IGNITING INNOVATION



Bob Foreman  
Software Engineering Lead  
LexisNexis Risk Solutions

Sleep Well with ECL:  
Job Automation and Scheduling



# Welcome!



## Today's Agenda:

- This Workshop teaches the workunit scheduling and process automation aspects of the ECL programming language. These are tools that allow you to standardize and optimize when/how a given set of tasks can be most efficiently accomplished within your production environment.
- We will start by introducing the job scheduling syntax in ECL which teaches you how to launch a workunit at a specific time or repetitively, based on time periods (such as, every hour, or every five minutes, or ...).
- We will also introduce the event-driven ECL syntax that allows you to launch workunits when a specific event occurs. It then applies these techniques to an extended real-world set of tools that automatically sprays files that appear on your Landing Zone and processes them to automatically add them to your existing production data.

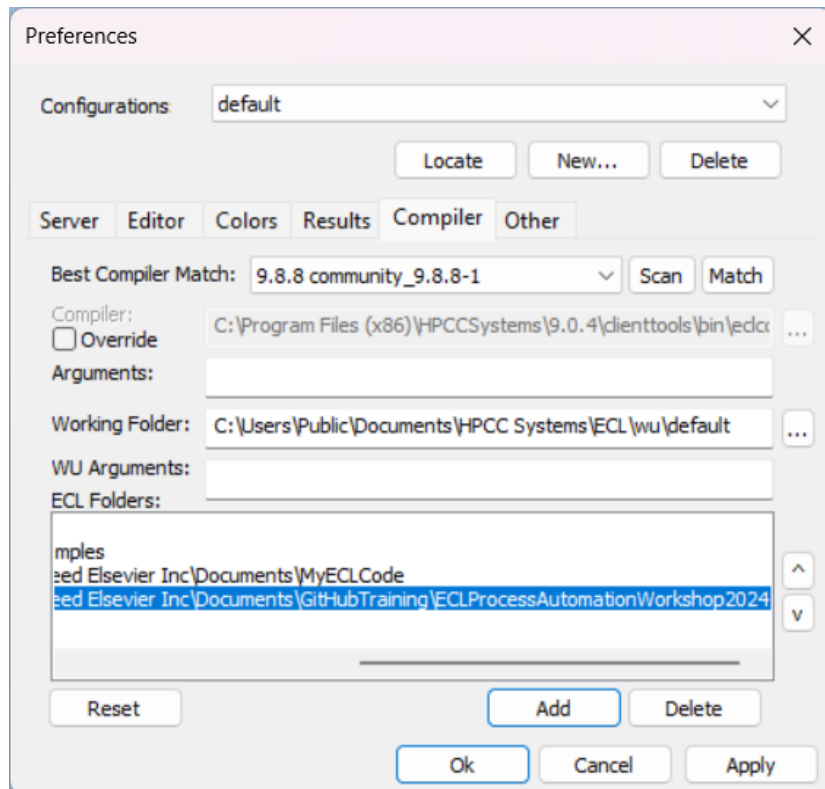
# Welcome!

## Follow along!

- The ECL code and docs for this workshop is contained in the following repository:

<https://github.com/hpcc-systems/ECLProcessAutomationWorkshop2024>

- Simply download or clone the repo then add that folder to your **ECL Folders** list on the **Preferences** page of your ECL IDE, or simply open it in your VSCode IDE.



# Introduction:



- Automating standard processes that can run without anyone's direct oversight is all about knowing “when” to launch the process. In ECL, that is handled by the **WHEN()** workflow service.
- The **WHEN()** workflow service is appended to any ECL Action (it must be an action, not a definition) that should delay its execution until the “appropriate time” that is either a specific time/day (using the **CRON()** function as its parameter), or whenever some specific named event occurs (using the **EVENT()** function as its parameter).

This workshop will show how to automate processes and schedule jobs in three stages:

1. **CRON()** Jobs - How to launch jobs that need to run periodically.
2. Triggering Events - How to launch jobs that need to run when something specific happens.
3. An Extensive Real-World Example - Production-ready code that uses these tools to monitor your Landing Zone for a specific file to show up, then automatically import and process new data files.



CRON Jobs

# CRON



## Overview:

The **CRON()** function takes a single string parameter that contains five space-delimited settings for

1. The minute
2. The hour
3. The day of month
4. The month
5. The day of week

Any omitted setting is indicated by an asterisk (\*).



# Organizing the base times: the *Every* MODULE



## Overview:

The *Every* MODULE structure contains a few standard settings to create strings that periodically trigger the CRON() function:

```
EXPORT Every(UNSIGNED1 n) := MODULE
  EXPORT DayAt    := '0 ' + n + ' * * *';
  EXPORT Hours    := '0 0-23/' + n + ' * * *';
  EXPORT Minutes  := '0-59/' + n + ' * * * *';
END;
```

Usage:

```
Every(2).DayAt    //Every Day at 2 A.M.
Every(3).Hours    //Every 3 hours
Every(15).Minutes //Every 15 minutes
```

# Track your events – The LOGOUT function



## Overview:

The LOGOUT function was written to track events in the workshop examples, writing these events to the ECL Agent Log :

```
IMPORT Std;
EXPORT LogOut(STRING s) := FUNCTION
  LogRec := RECORD
    STRING50 Comment;
    UNSIGNED4 Date := Std.Date.Today();
    UNSIGNED3 Time := Std.Date.CurrentTime();
  END;
  ds := DATASET([s], LogRec);
  LogStr := TRIM(s) + ' ' + ds[1].Date + '-' + ds[1].Time;
  AgtLog := Std.System.Log.dbglog(LogStr);
  RETURN WHEN(ds, AgtLog);
END;
```



# Testing your CRON Jobs



## Overview:

A simple BWR file of three (3) OUTPUT statements sending a LOGOUT message when a CRON event is detected :

```
IMPORT Std;  
//every two minutes:  
OUTPUT($.LogOut('2 minute run'),NAMED('TwoMinuteRun'),EXTEND): WHEN(CRON($.Every(2).Minutes));  
  
//on the hour, every hour:  
OUTPUT($.LogOut('Hourly run'),NAMED('HourRun'),EXTEND): WHEN(CRON($.Every(1).Hours));  
  
//daily at noon (UTC time)  
OUTPUT($.LogOut('Noon run'),NAMED('NoonRun'),EXTEND): WHEN(CRON($.Every(12).DayAt));
```



## Triggering Events

# Overview



- Just triggering a job at a certain time and logging that time is useful for scheduling jobs that need to run periodically, but it doesn't cover every need in automating your processes.
- You also need to be able to automatically launch jobs that do not have a regular period, that are the “on demand” type of processes.
- This means that we will now cover how you can launch jobs by triggering specific events from within your ECL code, using the NOTIFY() function.

# Using NOTIFY



- This example introduces the use of the NOTIFY() action to trigger a specific event, and the ORDERED() action to both launch the work and log it.
- To demonstrate that, we will set up a CRON job that fires once each minute, alternately triggering different events to launch one job on the odd- numbered minutes and another on the even-numbered ones

# Welcome!



## BWR\_NotifyJobs:

```
#WORKUNIT('name','Job Notify');
IMPORT Std;
//job set up
Events := ['LaunchEven','LaunchOdd']; //Event names
Time    := Std.Date.CurrentTime();
Minute  := Std.Date.Minute(Time);
MyEvent := Events[Minute % 2 + 1];    //even or odd
WorkTxt := ' was Triggered on ' + Std.Date.Today() + ' at ' + Time;
//do the work when the event triggers
OUTPUT('EVEN ' + WorkTxt) : WHEN(EVENT(Events[1],'*'));
OUTPUT('ODD ' + WorkTxt)  : WHEN(EVENT(Events[2],'*'));
//trigger and log the events
LaunchJob := NOTIFY(MyEvent,'*');
LogText   := 'NOTIFY() Pushed Event: ' + MyEvent;
LogEvent  := OUTPUT($.LogOut(LogText), NAMED('NotifyRun'),EXTEND);
ORDERED(LaunchJob,LogEvent) : WHEN(CRON($.Every(1).Minutes));
```

# Using EVENTEXTRA



- This example demonstrates how to pass additional information to the job that can be different for every instance launched using the EVENTEXTRA() function.

## BWR\_JobLaunch:

```
#WORKUNIT('name', 'Job Launch');
IMPORT Std;

//job set up
Events := ['LaunchEven', 'LaunchOdd']; //Event names
Time := Std.Date.CurrentTime();
Minute := Std.Date.Minute(Time);
MyEvent := Events[Minute % 2 + 1]; //even or odd

//do the work when the event triggers
Info := ['EVEN', 'ODD'];
Extra := '<EvenOdd>' + Info[Minute % 2 + 1] + '</EvenOdd>';
EventEx := '<Event>' + Extra + '</Event>';

Worktexts := EVENTEXTRA('EvenOdd') + ' was Triggered at: ' + Std.Date.Today() + '-' + Time;

OUTPUT(WorkTxt) : WHEN(EVENT(Events[1], '*'));
OUTPUT(WorkTxt) : WHEN(EVENT(Events[2], '*'));

//trigger and log the events
LaunchJob := NOTIFY(MyEvent, EventEx);
LogText := 'NOTIFY() Pushed Event: ' + MyEvent;
LogEvent := OUTPUT($.LogOut(LogText), NAMED('NotifyRunExtra'), EXTEND);
ORDERED(LaunchJob, LogEvent) : WHEN(CRON($.Every(1).Minutes));
```



Real World Example



# Introduction



## The Plan:

- Now that we've demonstrated all the key pieces to this type of process automation, we will expand on that with some real-world examples. We will create a tool to automatically spray files to your HPCC Systems whenever they appear on your environment's Landing Zone.
- After that, we'll use that tool to demonstrate spraying JSON, then XML, and then free-form text files. Once sprayed, these files are automatically processed in a standard manner, so they are ready to work with.

# The Design



- Data files come in periodically that need to be automatically processed in a standard manner whenever they show up, without human intervention. They come in pre-defined formats, as a batch (one or more data files). This code is based on code that was put into Production use over eight years ago (as of this writing). This is the design overview:
- A pre-defined “semaphore” file arrives on the Landing Zone containing a list of files to automatically spray and process. Those files must have already been transmitted to that same Landing Zone before the “semaphore” file arrives.
- A DFU workunit monitors for the arrival of that “semaphore” file, checking by filename for its appearance every fifteen minutes. When the file appears, it pushes an event to launch the spray process.

# The Design (Continued)



- An ECL automatic spray/process workunit is notified of the event pushed by the DFU workunit. It then reads the “semaphore” file, spraying each file listed in turn.
- As each file is sprayed, additional events (if any) are automatically pushed to further process the just-sprayed file.
- Once all files have been sprayed, the “semaphore” file is deleted from the Landing Zone so that another same-named file can eventually replace it, nominating a new set of files to spray.

# The Interface



- Because there will be many variables that will be unique to each specific set of files to spray and process, the first thing we need is an INTERFACE structure to contain them all. That way we can provide a concrete instance of the INTERFACE as a MODULE structure to specify only those options that need to be overridden from the default values in the INTERFACE.

**i\_AutoSprayFiles.ECL**

# The Interface



## The LZ Specs

```
//LZ specs
EXPORT STRING IP           := '';
EXPORT STRING LZpath       := '';
```

- The **IP** definition specifies the IP address of the environment's ESP server (also the host server for the environment's ECL Watch), which is typically where the Landing Zone is hosted. The default is an empty string, so you will always need to override this definition in the concrete instance of this INTERFACE that you use for production code. Barring that, you could instead just edit this definition to your environment's IP if you only have one.
- The **LZpath** definition specifies the path on the IP server to the actual Landing Zone. This default is also an empty string, so you will always need to override it or edit it to default to your environment's Landing Zone path (if you only have one).



## Semaphore File Specs

```
//semaphore file specs
EXPORT STRING FlagFile      := '';
EXPORT STRING FlagFilePath  := TRIM(LZpath) + FlagFile;
EXPORT STRING FileToRead    := IF(FlagFilePath[1]='/', Std.Str.FindReplace(FlagFilePath[2..], '/', '::' ),
                                Std.Str.FindReplace(FlagFilePath, '/', '::' ));
```

- The **FlagFile** definition names the semaphore file. You will always need to override this in your concrete instance.
- The **FlagFilePath** definition concatenates the semaphore file name to the LZpath. You will always need to override this in your concrete instance.
- The **FileToRead** definition translates the slash ('/') directory delimiters to the double colons ('::') that the DFU uses so it can find the file. This default expression should only be overridden if you absolutely need to, since it already covers what is needed in most concrete instances.

# The Interface



## Spray (Import) Target Specs

```
//spray target specs  
EXPORT STRING SprayTargetDir := '';  
EXPORT STRING TargetCluster  := IF(__CONTAINERIZED__, 'data', Std.System.ThorLib.Group());
```

- The **SprayTargetDir** definition names the directory to spray to on the cluster. You will always need to override this in your concrete instance. This is analogous to the Target Scope setting you provide when manually spraying files using ECL Watch.
- The **TargetCluster** definition names the target for the spray. Because this takes a different form on Kubernetes clusters, the default value uses the `__CONTAINERIZED__` compile time constant to make the default value an appropriate one for Kubernetes or bare metal platforms. You should only override this in your concrete instance if your data plane name in your Kubernetes environment is not “data.” If you have a bare metal environment, the **Std.System.ThorLib.Group()** function should provide the correct name.



# The Interface



## The Events

```
EXPORT STRING NewFileEvent    := '';  
EXPORT STRING EventToPush     := '';  
EXPORT STRING EventToLaunch   := '';
```

- The **NewFileEvent** definition names the event that instigates the spray when the semaphore file has appeared on the Landing Zone. You should always override this in your concrete instance.
- The **EventToPush** definition names the event that instigates the processing of each individual file, once it has been sprayed. You only need to override this in your concrete instance if you are automatically processing each file as they are sprayed.
- The **EventToLaunch** definition names the event that instigates the processing of all the files, after they have all been sprayed and processed. You only need to override this in your concrete instance if you are automatically processing all files after they have been sprayed.

# The Interface



## File Type Flag

```
EXPORT UNSIGNED1 SprayType    := 0;
```

- The SprayType definition the file type of all files listed in the semaphore file. You must always override this in your concrete instance. Valid values are: 1=Delimited, 2=Fixed, 3=XML, 4=JSON.

# The Interface



## CSV Spray (Import) Options

```
EXPORT STRING FieldDelimiter := '\\,';  
EXPORT STRING RecDelimiter   := '\\n,\\r\\n';  
EXPORT STRING QuoteDelimiter := '\"';  
EXPORT STRING EscapeChar     := '\\';
```

- The **FieldDelimiter** definition specifies the delimiter between each field. You should only override this in your concrete instance if your field delimiter is not a comma.
- The **RecDelimiter** definition specifies the delimiter between each record. You should only override this in your concrete instance if your record delimiter is not either the Unix-style newline character (`\n`) or DOS-style carriage return/line feed (`\r\n`).
- The **QuoteDelimiter** definition specifies the quote characters used around string data that may contain field or record delimiter characters as part of their data. You should only override this in your concrete instance if your quote character is not a double-quote (`"`).
- The **EscapeChar** definition specifies the character that indicates a special character to include in the data. You should only override this in your concrete instance if your escape character is not a backslash (`\`).

# The Interface



## Spray (Import) options for all files

```
EXPORT BOOLEAN OverwriteOK    := TRUE;  
EXPORT BOOLEAN Replicate      := TRUE;  
EXPORT BOOLEAN CompressData   := FALSE;  
EXPORT UNSIGNED1 PauseSecs    := 0;
```

- The **OverwriteOK** definition specifies whether to allow the spray to overwrite an existing file with the same name and path. You should only override this in your concrete instance if you do not want to allow overwrites.
- The **Replicate** definition specifies whether to spray each file part and also replicate each part on the appropriate adjacent nodes during the spray process. You should only override this in your concrete instance if your environment is setup for automatic delayed replication.
- The **CompressData** definition specifies whether to automatically LZW compress the data as it is being sprayed to the cluster. You should only override this in your concrete instance if you want to always compress the sprayed files.
- The **PauseSecs** definition specifies how long (in seconds) to pause between spraying each file listed in the semaphore file to allow the processing workunit (launched by the **EventToPush** event) to finish. This can prevent “fast” sprays from overloading “slower” processing jobs. You should only override this in your concrete instance if you have tested your process and know that your automatic process job consistently takes longer than your spray, which could cause problems if the subsequent sprays are not delayed, thus allowing time for the previous file’s processing to complete.

# The Events



## ModEvents.ecl

```
EXPORT Mod_Events := MODULE, VIRTUAL
EXPORT STRING Who := 'x';
EXPORT STRING Spray := Who + '_Spray';
EXPORT STRING Process := Who + '_Process';
EXPORT STRING Launch := Who + '_Launch';
EXPORT STRING Final := Who + '_Final';
END;
```

- Since this AutoSpray automation is an event-driven process, we need to define some events.
- This **Mod\_Events** MODULE is declared as VIRTUAL because it is used only as the template for each concrete MODULE instance that inherits it. It only defines the most common requirements that all automated spray processes will require.

# The Events



## ModEvents.ecl

```
EXPORT Mod_Events := MODULE, VIRTUAL
EXPORT STRING Who   := 'x';
EXPORT STRING Spray := Who + '_Spray';
EXPORT STRING Process := Who + '_Process';
EXPORT STRING Launch := Who + '_Launch';
EXPORT STRING Final  := Who + '_Final';
END;
```

- The **Who** definition should always be overridden in your concrete events MODULE that inherits this one. Doing that allows you to make the actual events for each real-world implementation unique to that one set of processes. That way, you can have as many implementations of AutoSpray active and operating in your environment as you need, all at the same time.
- The **Spray**, **Process**, **Launch**, and **Final** definitions all concatenate the **Who** definition's unique value to standard constants that make it easy to see which workunits are monitoring for which specific events. This will all become clearer when we create several implementations of all this, later.

# The Utilities



## Mod\_Utills.ecl

- There are three standard functions I created for this process that will be useful in both the core code and each implementation. We organized them into a single MODULE structure.
- **RawFileName()** FUNCTION - parse a standard path/filename from the DFU (which uses double-colons as the directory delimiter) to extract just the filename portion.
- **CapCaret()** FUNCTION - parses a full path and returns a string with a caret character (^) prepended to each uppercase letter.
- **Pause()** FUNCTION - an ECL “wrapper” for the C++ sleep() function.



# The Utilities



## RawFileName()

```
EXPORT RawFilename(String filename) := FUNCTION
  Splits := Std.Str.SplitWords(filename, '::');
  SplitCnt := COUNT(Splits);
  RETURN Splits[SplitCnt];
END;
```

- The **RawFileName()** function starts by using the **Std.Str.SplitWords()** standard library function to parse the passed-in filename argument into individual elements in the **Splits** set. It uses the double-colon directory delimiter (::) as the indicator for how to split each element into a separate “word” in the set (without including the double-colons).
- The **SplitCnt** definition then uses the **COUNT()** function to determine the number of elements in the set. That allows the function to **RETURN** the last element in the **Splits** set (the **SplitCnt** element). That last element only contains the filename without all the leading path text that came in as the input filename parameter.

# The Utilities



## CapCaret()

```
EXPORT CapCaret(String filename) := FUNCTION
  SetU := ['A','B','C','D','E','F','G','H','I','J','K','L','M',
           'N','O','P','Q','R','S','T','U','V','W','X','Y','Z'];
  chars := LENGTH(filename);
  ds := DATASET(chars, TRANSFORM({STRING char},
                                s := filename[COUNTER];
                                SELF.char := IF(s IN SetU, '^' + s, s)));
  Cfile := ROLLUP(ds, TRUE, TRANSFORM({STRING char},
                                       SELF.char := LEFT.char + RIGHT.char));
  RETURN Cfile[1].char;
END;
```

- The function begins with the **SetU** definition that provides a set of all the possible uppercase letters. I only needed support for the English language, so if you're working with other languages you may need to edit this set.

# The Utilities



## CapCaret()

```
EXPORT CapCaret(String filename) := FUNCTION
  SetU := ['A','B','C','D','E','F','G','H','I','J','K','L','M',
           'N','O','P','Q','R','S','T','U','V','W','X','Y','Z'];
  chars := LENGTH(filename);
  ds := DATASET(chars, TRANSFORM({STRING char},
                                s := filename[COUNTER];
                                SELF.char := IF(s IN SetU, '^' + s, s)));
  Cfile := ROLLUP(ds, TRUE, TRANSFORM({STRING char},
                                       SELF.char := LEFT.char + RIGHT.char));
  RETURN Cfile[1].char;
END;
```

- The **chars** definition provides the number of characters in the passed *filename* argument.

# The Utilities



## CapCaret()

```
EXPORT CapCaret(String filename) := FUNCTION
  SetU := ['A','B','C','D','E','F','G','H','I','J','K','L','M',
           'N','O','P','Q','R','S','T','U','V','W','X','Y','Z'];
  chars := LENGTH(filename);
  ds := DATASET(chars, TRANSFORM({STRING char},
                                s := filename[COUNTER];
                                SELF.char := IF(s IN SetU, '^' + s, s)));
  Cfile := ROLLUP(ds, TRUE, TRANSFORM({STRING char},
                                       SELF.char := LEFT.char + RIGHT.char));
  RETURN Cfile[1].char;
END;
```

- The **ds** DATASET uses that **chars** value to create a separate record for each original letter, prepending each uppercase letter with the caret character (^). The **char** field is a variable-length STRING type so it can contain as many characters as needed (in this case, either one or two).

# The Utilities



## CapCaret()

```
EXPORT CapCaret(String filename) := FUNCTION
  SetU := ['A','B','C','D','E','F','G','H','I','J','K','L','M',
           'N','O','P','Q','R','S','T','U','V','W','X','Y','Z'];
  chars := LENGTH(filename);
  ds := DATASET(chars, TRANSFORM({STRING char},
                                s := filename[COUNTER];
                                SELF.char := IF(s IN SetU, '^' + s, s)));
  Cfile := ROLLUP(ds, TRUE, TRANSFORM({STRING char},
                                       SELF.char := LEFT.char + RIGHT.char));
  RETURN Cfile[1].char;
END;
```

- The **Cfile** definition uses the **ROLLUP()** function to de-duplicate the **ds** records using the **TRUE** keyword as its record-matching criteria, which forces every record to match every other record. This results in a single-record record set with all the concatenated char field values in that single record.

# The Utilities



## CapCaret()

```
EXPORT CapCaret(String filename) := FUNCTION
  SetU := ['A','B','C','D','E','F','G','H','I','J','K','L','M',
           'N','O','P','Q','R','S','T','U','V','W','X','Y','Z'];
  chars := LENGTH(filename);
  ds := DATASET(chars, TRANSFORM({STRING char},
                                s := filename[COUNTER];
                                SELF.char := IF(s IN SetU, '^' + s, s)));
  Cfile := ROLLUP(ds, TRUE, TRANSFORM({STRING char},
                                       SELF.char := LEFT.char + RIGHT.char));
  RETURN Cfile[1].char;
END;
```

- Finally, the **RETURN** expression returns that single string char field value. So, if the passed filename contains `~RTTEST::IN::Myfile.csv`, then the returned string from the function will be `~^R^T^T^E^S^T::^I^N::^Myfile.csv` which will allow the Linux file system to actually find the file.

# The Utilities



## Pause()

```
EXPORT Pause(secs) := BEGINC++  
    sleep(secs);  
ENDC++;
```

- Sometimes, the file spray goes quickly, while the subsequent process job for that file can take longer to run. Because that process job is in a separate workunit, which must complete before it can go back to waiting for the next instance of the event that triggers it, this can cause it to miss launch events for subsequent files (i.e., not execute on all the files). In that case, you can use this **Pause()** function to delay the start of the next spray by the number of secs (seconds) you specify, thereby giving the process job time to complete. How long exactly to delay will have to be determined by your own experimentation, but we will be using this in one of the final examples for this workshop.





Taking it to the Streets  
Building the AutoSpray Tool

# Converting Data to Spray



- **MOD\_AutoSprayFiles:**

- The real work of spraying the files and launching events is all contained in a single MODULE structure.
- The Mod\_AutoSprayFiles MODULE structure begins with a standard IMPORT declaration for the current directory (\$) and the ECL Standard Library (Std).

```
IMPORT $,Std;  
EXPORT Mod_AutoSprayFiles($.I_AutoSprayFiles P) := MODULE
```

- The single parameter is required and must be a concrete instance of the I\_AutoSprayFiles INTERFACE we previously defined. That passed parameter is named P -- normally we use more verbose parameter names, but since this will be referenced many times in the rest of the code in this MODULE, we chose brevity over verbosity.

# Inside Mod\_AutoSprayFiles



## The Semaphore (Messaging) File:

```
LZfileRec := RECORD
  STRING200 fname;
  UNSIGNED4 size;
  STRING    tag;
END;
```

```
LZfname      := '~file::' + $.Mod_Utils.CapCaret(P.IP) + '::' + $.Mod_Utils.CapCaret(P.FileToRead);
SHARED LZds   := DATASET(LZfname,LZfileRec,CSV);
SHARED FileCnt := COUNT(LZds);
SHARED MaxFiles := 4;
```

- The **LZfname** definition creates the full path to the semaphore file from the **P.IP** and **P.FileToRead** values, pass each through the **CapCaret()** function to ensure that Linux can find the correct file. This is necessary because the file is on the Landing Zone, which is a Linux box on our Docker Desktop Kubernetes environment.

# Inside Mod\_AutoSprayFiles



## The Semaphore (Messaging) File:

```
LZfileRec := RECORD
  STRING200 fname;
  UNSIGNED4 size;
  STRING    tag;
END;
```

```
LZfname      := '~file::' + $.Mod_Utils.CapCaret(P.IP) + '::' + $.Mod_Utils.CapCaret(P.FileToRead);
SHARED LZds   := DATASET(LZfname, LZfileRec, CSV);
SHARED FileCnt := COUNT(LZds);
SHARED MaxFiles := 4;
```

- The **LZds** DATASET definition declares the Semaphore file for use in the rest of our ECL code. The **FileCnt** definition determines the number of records in the semaphore file, and the **MaxFiles** definition is a constant that specifies the maximum number of files that can be sprayed as a single batch (I'll cover why this is later, when we get to the code that uses this).

# Inside Mod\_AutoSprayFiles



## Some Prep Work

```
//Create the target file name:
SHARED FullFile(STRING fil) := TRIM(P.SprayTargetDir) + fil;

//Log the file sprays:
SHARED SprayMsg(STRING str) := OUTPUT(DATASET([{str}],{STRING line}), NAMED('SprayProgress'),EXTEND);

//clean up after all files sprayed:
SHARED DeleteSemaphore := Std.File.DeleteExternalFile(P.IP, P.FlagFilePath);
```

- The **FullFile()** function creates the full DFU path and filename for each sprayed file. The **SprayMsg()** action writes a log entry to the *SprayProgress* result for the workunit that sprays the files. The **DeleteSemaphore** definition removes the Semaphore file from the Landing Zone after all files have been sprayed. This sets up the wait for another appearance of that file to initiate spraying another batch of files.

# Inside Mod\_AutoSprayFiles



## More Prep Work:

```
//TimeStamp:  
SHARED TimeStamp := Std.Date.Today() + ' ' + Std.Date.CurrentTime();  
  
//Log the events pushed:  
SHARED EventMsg(String str) := Std.System.Log.addWorkunitInformation(str);
```

- The **TimeStamp** definition creates a string containing the current date and time in *YYYYMMDD HHMMSS* format.
- The **EventMsg()** function adds the passed str argument to the workunit's Info(s) queue, which displays at the bottom of the workunit's summary page in ECL Watch.

# Inside Mod\_AutoSprayFiles



## The Semaphore File Monitor:

```
EXPORT WaitForFile := Std.File.MonitorFile(P.NewFileEvent, P.IP, P.FlagFilePath, TRUE,-1);
```

- The **WaitForFile** definition is the EXPORT definition you use in production code to initiate the entire automatic file spray process. It uses the **Std.File.MonitorFile** Standard Library action to launch a DFU workunit that checks the Landing Zone periodically for your Semaphore file. When the file appears, it pushes the *P.NewFileEvent* to the environment, which the workunit waiting for that event will “notice” and start its work.
- NOTE: The default monitor period is 900 seconds (15 minutes). This is the DFU Server's *monitorinterval* setting. That setting can be found in the **environment.xml** file for bare metal configurations, and in **values.yaml** for Kubernetes configurations.

# Inside Mod\_AutoSprayFiles – 4 Spray Functions



## The VarSpray() function:

```
EXPORT VarSpray(STRING TheFile,UNSIGNED4 RecSize) :=  
Std.File.SprayDelimited(P.IP,TRIM(P.LZpath)+TheFile, IF(RecSize>0,RecSize,8192),  
                        P.FieldDelimiter, P.RecDelimiter, P QuoteDelimiter, P.TargetCluster,  
                        FullFile(TheFile),-1,,,P.OverwriteOK, P.Replicate, P.CompressData,P.EscapeChar);
```

- The **VarSpray()** function takes two parameters: **TheFile** is the full path to the file to spray, and the **RecSize** is the maximum length of any record. This is the function that sprays any type of Delimited file.
- It uses the **Std.File.SprayDelimited()** function from the ECL Standard Library to execute the spray operation, passing it parameters from the **P** concrete instance of the **I\_AutoSprayFiles** INTERACE passed to the **Mod\_AutoSprayFiles** MODULE from the code that calls it (just as all the spray functions do). That concrete instance MODULE structure that is passed in is where you specify exactly which characters are used for your field and record delimiters, allowing you to work with any Delimited file format.



# Inside Mod\_AutoSprayFiles – 4 Spray Functions



## The FixSpray() function:

```
EXPORT FixSpray(String TheFile, UNSIGNED4 RecSize) :=  
    Std.File.SprayFixed(P.IP, TRIM(P.LZpath) + TheFile, RecSize,  
        P.TargetCluster, FullFile(TheFile), -1, P.OverwriteOK, P.Replicate, P.CompressData);
```

- The **FixSpray()** function takes two parameters: **TheFile** is the full path to the file to spray, and the **RecSize** is the actual length of every record. This is the function that sprays any type of file whose records are fixed number of bytes.
- It uses the **Std.File.SprayFixed()** function from the ECL Standard Library to execute the spray operation. This function handles any type of file whose records are all a fixed length, whether that data is all text or binary.

# Inside Mod\_AutoSprayFiles – 4 Spray Functions



## The XMLSpray() function:

```
EXPORT XmlSpray(String TheFile, UNSIGNED4 RecSize, STRING rowtag) :=  
    Std.File.SprayXML(P.IP, TRIM(P.LZpath) + TheFile,  
    IF(RecSize>0, RecSize, 8192), rowtag, P.TargetCluster, FullFile(TheFile),  
    -1, P.OverwriteOK, P.Replicate, P.CompressData);
```

- The **XmlSpray()** function takes three parameters: **TheFile** is the full path to the file to spray, the **RecSize** is the maximum length of any record, and the **rowtag** is the tag name that delimits each record. This is the function that sprays well-structured XML data files, where each record is delimited by a set of rowtag tags.
- It uses the **Std.File.SprayXML()** function from the ECL Standard Library to execute the spray operation.

# Inside Mod\_AutoSprayFiles – 4 Spray Functions



## The JsnSpray() function:

```
EXPORT JsnSpray(String TheFile, UNSIGNED4 RecSize, STRING rowtag) :=  
    Std.File.SprayJSON(P.IP, TRIM(P.LZpath) + TheFile,  
    IF(RecSize>0, RecSize, 8192), rowtag,, P.TargetCluster, FullFile(TheFile),  
    -1,,, P.OverwriteOK, P.Replicate, P.CompressData);
```

- The **JsnSpray()** function takes three parameters: **TheFile** is the full path to the file to spray, the **RecSize** is the maximum length of any record, and the **rowtag** is the tag name that delimits the set of records. This is the function that sprays well-structured JSON data files, where each record is the set of records contained within the rowtag tag.
- It uses the **Std.File.SprayJSON()** function from the ECL Standard Library to execute the spray operation.

# Inside Mod\_AutoSprayFiles – Actions



## Spray(Import) and Process Actions – EXTRAEVENT():

```
FileEx(STRING fil) := '<FileName>' + FullFile(Fil) + '</FileName>';  
EventEx(STRING Ex) := '<Event>' + FileEx(Ex) + '</Event>';
```

- The only way to pass information about a specific instance of an event handling process is in an XML string. That XML string must be in this format:  
**'<Event><mytag>Data to Pass</mytag></Event>'**
- The **EVENTEXTRA()** function requires the actual information in the string to be contained within the required **<Event></Event>** tags, nested within other tags that you create (represented here as **<mytag></mytag>**) so that it can parse the XML with.
- The **EventEx()** and **FileEx()** functions work together to create the XML string for each sprayed file to pass the name of the current file to act upon to the process code. The result will look like this:  
**'<Event><FileName>myfile</FileName></Event>'**

# Inside Mod\_AutoSprayFiles – Actions



## Action to process each file:

```
PushEvent(STRING TheFile) := ORDERED(NOTIFY(P.EventToPush,EventEx(TheFile)),  
                                     EventMsg(TimeStamp + ' Event ' + P.EventToPush + ' pushed for '+ TheFile));
```

- The **PushEvent** function does two things in sequence, using the **ORDERED** action to specify the execution order:
  1. The **NOTIFY()** function triggers the event that your process workunit is waiting for, passing the name of the current file to process (from **TheFile** argument) in the **NOTIFY()** function's second parameter.
  2. The **EventMsg()** function posts an Information message to the workunit noting the time and event that was triggered for **TheFile**.

# Inside Mod\_AutoSprayFiles – Actions



## Action to trigger the post-spray final event:

```
PushFinalEvent(STRING TheFile) := ORDERED(OUTPUT(LZds,,FullFile(TheFile),CSV,OVERWRITE),  
                                           NOTIFY(P.EventToLaunch,EventEx(TheFile)),  
                                           EventMsg(TimeStamp + ' Event ' + P.EventToLaunch + ' pushed for ' + TheFile));
```

- The **PushFinalEvent** function does three things in sequence, using the **ORDERED** action to specify the execution order:
  1. The **OUTPUT** action writes a copy of the Semaphore file (**LZds**) to the spray target directory, overwriting any previous version that may have been there. This saves it for use by the workunit that's waiting for the **P.EventToLaunch** to commence its work.
  2. The **NOTIFY()** function triggers the event that your final workunit is waiting for, passing the name of the Semaphore file (from **TheFile** argument) to process in the **NOTIFY()** function's second parameter.
  3. The **EventMsg()** function posts an Information message to the workunit noting the time and event that was triggered for **TheFile**.

# Inside Mod\_AutoSprayFiles – Actions



## Action to spray (import) a file in the semaphore file:

```
Msg1(n) := TimeStamp + ' - Spraying File: ' + LZds[n].fname;  
Msg2(n) := TimeStamp + ' NO SPRAY TYPE SPECIFIED- ' + LZds[n].fname;  
SprayOne(UNSIGNED1 num) := IF(num <= FileCnt AND LZds[num].fname <> '',  
    ORDERED(SprayMsg(Msg1(num)),  
        CASE(P.SprayType, 1 => VarSpray(LZds[num].fname, LZds[num].size),  
                2 => FixSpray(LZds[num].fname, LZds[num].size),  
                3 => XmlSpray(LZds[num].fname, LZds[num].size, LZds[num].tag),  
                4 => JsnSpray(LZds[num].fname, LZds[num].size, LZds[num].tag),  
        SprayMsg(Msg2(num))),  
    IF(P.SprayType IN [1,2,3,4] AND P.EventToPush <> '',  
        ORDERED(PushEvent(LZds[num].fname),  
            $.Mod_Utils.Pause(P.PauseSecs))  
    ));
```

- The **Msg1()** and **Msg2()** functions are defined separately from where they're used just for better organization.
- The **SprayOne()** function accomplishes the actual spray operation. It uses the **IF()** function to detect whether there is a num entry in the Semaphore file that contains the name of a file to spray. If not, it does nothing—a False expression is not required when the True expression is an action.

# Inside Mod\_AutoSprayFiles – Actions



## Action to spray (import) a file in the semaphore file:

```
Msg1(n) := TimeStamp + ' - Spraying File: ' + LZds[n].fname;
Msg2(n) := TimeStamp + ' NO SPRAY TYPE SPECIFIED- ' + LZds[n].fname;
SprayOne(UNSIGNED1 num) := IF(num <= FileCnt AND LZds[num].fname <> '',
    ORDERED(SprayMsg(Msg1(num))),
    CASE(P.SprayType, 1 => VarSpray(LZds[num].fname, LZds[num].size),
    2 => FixSpray(LZds[num].fname, LZds[num].size),
    3 => XmlSpray(LZds[num].fname, LZds[num].size, LZds[num].tag),
    4 => JsnSpray(LZds[num].fname, LZds[num].size, LZds[num].tag),
    SprayMsg(Msg2(num))),
    IF(P.SprayType IN [1,2,3,4] AND P.EventToPush <> '',
    ORDERED(PushEvent(LZds[num].fname),
    $.Mod_Utils.Pause(P.PauseSecs))
    ));
```

- Only if there is a filename to spray does it do three things in sequence, using the ORDERED action to specify the execution order:
  1. The **SprayMsg()** function adds the **Msg1()** text for the current file to the *SprayProgress* log file for the workunit.



# Inside Mod\_AutoSprayFiles – Actions



## Action to spray (import) a file in the semaphore file:

```
Msg1(n) := TimeStamp + ' - Spraying File: ' + LZds[n].fname;
Msg2(n) := TimeStamp + ' NO SPRAY TYPE SPECIFIED-' + LZds[n].fname;
SprayOne(UNSIGNED1 num) := IF(num <= FileCnt AND LZds[num].fname <> '',
    ORDERED(SprayMsg(Msg1(num)),
        CASE(P.SprayType, 1 => VarSpray(LZds[num].fname, LZds[num].size),
            2 => FixSpray(LZds[num].fname, LZds[num].size),
            3 => XmlSpray(LZds[num].fname, LZds[num].size, LZds[num].tag),
            4 => JsnSpray(LZds[num].fname, LZds[num].size, LZds[num].tag),
            SprayMsg(Msg2(num))),
    IF(P.SprayType IN [1,2,3,4] AND P.EventToPush <> '',
        ORDERED(PushEvent(LZds[num].fname),
            $.Mod_Utils.Pause(P.PauseSecs))
    ));
```

2. The **CASE()** function calls the appropriate spray function, based on the **P.SprayType** value, or defaults to using the **SprayMsg()** function adding the **Msg2()** text to the *SprayProgress* log file for the workunit.
3. The **IF()** function detects whether there is a valid **P.SprayType** value and a non-blank **P.EventToPush**. If both are true, then it uses the **ORDERED** action to call the **PushEvent()** function to launch the process workunit, followed by the **\$.Mod\_Utils.Pause()** function that pauses for the number of seconds specified in **P.PauseSecs** (which does nothing if left to its default zero value).

# Inside Mod\_AutoSprayFiles – Actions



## Spray (import) all files in the semaphore file:

```
#DECLARE(SprayString);
#DECLARE(Ndx);
#SET(Ndx, 1);
#LOOP
  #IF(%Ndx% = 1)
    #SET(SprayString,'ORDERED(SprayOne(1)');
  #ELSEIF(%Ndx% > MaxFiles)
    #BREAK
  #ELSE
    #APPEND(SprayString,', SprayOne(' + '%Ndx%' + ')');
  #END
#SET(Ndx, %Ndx% + 1)
#END
#APPEND(SprayString,')');
SprayAllFiles := '%SprayString%';
```

- The **SprayAllFiles** definition expression is generated by the ECL Template Language. This allows us to set an optimal maximum number of files for each real-world instance of this code that we create.

- Given that the **MaxFiles** constant expression is four, this is the expression that is generated by the **%SprayString%** symbol:

ORDERED(SprayOne(1), SprayOne(2), SprayOne(3), SprayOne(4));

which becomes the expression for the **SprayAllFiles** definition.

# Inside Mod\_AutoSprayFiles – Actions



## Spray (import) all files in the semaphore file:

```
#DECLARE(SprayString);
#DECLARE(Ndx);
#SET(Ndx, 1);
#LOOP
  #IF(%Ndx% = 1)
    #SET(SprayString,'ORDERED(SprayOne(1)');
  #ELSEIF(%Ndx% > MaxFiles)
    #BREAK
  #ELSE
    #APPEND(SprayString,',SprayOne(' + '%Ndx'% + ')');
  #END
#SET(Ndx, %Ndx% + 1)
#END
#APPEND(SprayString,'');
SprayAllFiles := '%SprayString%';
```

- The **ORDERED** action calls the **SprayOne** action to spray up to four files, passing each record number in the Semaphore file to the **SprayOne** action. If you want to allow a different maximum number of files, you just change the **MaxFiles** definition value.

# Inside Mod\_AutoSprayFiles – Actions



## Spray (import) files then complete:

```
SprayThem := ORDERED(SprayAllFiles, IF(P.EventToLaunch <> '', PushFinalEvent(P.FlagFile)),  
                      SprayMsg(TimeStamp + ' - Deleting Semaphore'),  
                      DeleteSemaphore);
```

- The **SprayThem** action uses the **ORDERED** action to specify the execution order:
  1. Spray all the files with the **SprayAllFiles** action.
  2. Push the **P.EventToLaunch** (if not blank).
  3. The **SprayMsg()** function adds text to the *SprayProgress* log file for the workunit.
  4. The **DeleteSemaphore** action deletes the Semaphore file from the Landing Zone.

# Inside Mod\_AutoSprayFiles – Actions



## Decide to either spray (import) or fail for too many files in batch:

```
EXPORT SprayFiles :=  
  IF(FileCnt <= MaxFiles,  
    SprayThem,  
    ORDERED(EventMsg(TimeStamp + ' ' + P.FlagFilePath + ' --' +  
      ' TOO MANY FILES: ' + FileCnt + ' -- MAX FILES: ' + MaxFiles),  
      DeleteSemaphore));  
END;
```

- The **SprayFiles** action is the ***starting point for everything in this process***. It is the EXPORT action that initiates the spray process in your production code. It will either run the **SprayThem** action, or use the **EventMsg()** function to add the TOO MANY FILES text to the workunit Info list and delete the semaphore.



## Implementing the AutoSpray Tool XML Demo

# Generating Data to Spray(Import)



## GenSalesRec.ecl

- This code demonstrates techniques for generating test data and emulating working with files that contain additional data to add on a periodic basis.
- This code generates daily "sales" data for a 100-store chain of "dollar" stores where items are sold for 1-5 dollars. The result is a nested child dataset where each "store" record for a given date contains a child dataset containing a "sales" record for each SKU, showing how many were "sold" and the total revenue for that SKU for that day.
- Since both XML and JSON examples will use the same type of data, we created a single "record generator" to use for both (the actual data will be different in each).

# Generating Data to Spray(Import)



## GenSalesRec - The Basics:

```
IMPORT Std, $;
CapIt(String s) := $.Mod_Utils.CapCaret(s);
SKUrec := RECORD
  STRING5      SKU;           //item sold
  UNSIGNED2    NumSold;       //number sold
  UDECIMAL8    TotalAmt;      //total revenue
END;

rec := RECORD
  UNSIGNED2    StoreID;
  UNSIGNED4    TransDate;
  DATASET(SKUrec) Sales;
END;
```

- **CapIt()** redefines the **\$.Mod\_Utils.CapCaret()** function, just to simplify the code.
- The **SKUrec** and **Rec** RECORD structures define the layout of the nested child dataset that is generated.



# Generating Data to Spray(Import)



## GenSalesRec – More Basics

```
Ltrs      := 'ABCDEFGHIJKLMNOPQRSTUVWXYZ';  
Idx       := RANDOM() % 26 + 1;  
  
GenSKU    := Ltrs[Idx]+INTFORMAT(RANDOM()%1000,4,1);  
  
SKUcnt    := 500; //number of items for "sale"  
Stores    := 100;  
Days      := 3;  
RecCnt    := Stores * Days;
```

- The **GenSKU** definition uses the **Ltrs** and **Idx** to generate a “random” 5-character SKU to use as a “product” identifier, starting with an uppercase letter followed by a 4-digit number.
- The **SKUcnt** defines the number of “products” sold in each store. The **RecCnt** uses the **Stores** and **Days** to define the total number of parent records to generate (in this case, 300, since this is intended to create three days of sales for a 100-store chain).

# Generating Data to Spray(Import)



## GenSalesRec – Generate “SKUs” and their prices:

```
SKUs := FUNCTION
  ds := DATASET(SKUcnt + 100,
    TRANSFORM({STRING5 SKU, UNSIGNED1 Price},
      SELF.SKU := GenSKU,
      SELF.Price := RANDOM() % 5 + 1));
  dds := DEDUP(SORT(ds, SKU), SKU)[..SKUcnt];
  RETURN dds;
END      : INDEPENDENT; //do this once, only
SKU_DCT := DICTIONARY(SKUs, {SKU => price});
SKUprice(STRING8 s) := SKU_DCT[s].price;
```

- It does this with the ds DATASET that generates SKUcnt + 100 records with random SKU field values generated by the GenSKU definition and random Price field values (between 1 and 5). The SORT() and DEDUP() functions remove any duplicate SKU value records, then the square brackets eliminate the extra records so the function only returns the correct number.

- The **SKUs** function takes no parameters because it always generates the number of SKU records defined by **SKUcnt**.

# Generating Data to Spray(Import)



## GenSalesRec – Generate “SKUs” and their prices:

```
SKUs := FUNCTION
  ds := DATASET(SKUcnt + 100,
    TRANSFORM({STRING5 SKU, UNSIGNED1 Price},
      SELF.SKU := GenSKU,
      SELF.Price := RANDOM() % 5 + 1));
  dds := DEDUP(SORT(ds, SKU), SKU)[..SKUcnt];
  RETURN dds;
END : INDEPENDENT; //do this once, only
SKU_DCT := DICTIONARY(SKUs, {SKU => price});
SKUprice(STRING8 s) := SKU_DCT[s].price;
```

- The **INDEPENDENT** workflow service appended to the **FUNCTION** definition ensures that the record set of SKUs is generated globally and only once for the workunit so that all code for the rest of the workunit uses the same set of data records.
- The **SKU\_DCT DICTIONARY** definition provides an index into the SKUs by the SKU value and returns the price. Then the **SKUprice()** function uses that SKU\_DCT to return the price for a particular SKU.

# Generating Data to Spray(Import)



## GenSalesRec – Generate “sales” for each SKU:

```
Sales :=  
    DATASET(RecCnt * SKUcnt,  
            TRANSFORM({UNSIGNED4 ID, UNSIGNED4 C},  
                      SELF.ID := COUNTER,  
                      SELF.C := RANDOM() % SKUcnt))  
    : INDEPENDENT; //do this once, only
```

```
SalesDCT := DICTIONARY(Sales, {ID => C});  
SalesCnt(UNSIGNED4 i) := SalesDCT[i].C;
```

- The **SalesDCT DICTIONARY** definition provides an index into the Sales by the ID value and returns the C value. Then the **SalesCnt()** function uses that SalesDCT to return the C value for a particular ID. This function is used to populate the actual “sales” numbers for each SKU each day.

The **Sales** definition generates the total number of “Sales” records for all the generated data, as defined by **RecCnt \* SKUcnt**. It does this with a DATASET that generates records with unique ID field values generated by the COUNTER and random C field values (between 1 and SKUcnt) that will represent the number of “sales” in one day for a particular SKU.

# Generating Data to Spray(Import)



## GenSalesRec – Generate “sales” dates:

```
SET OF UNSIGNED4 dates := FUNCTION
  GD      := Std.Date.Today();
  JD      := Std.Date.FromGregorianCalendar(GD);
  C2GD(C) := Std.Date.ToGregorianCalendar(JD-C);
  ds      := DATASET(Days-1,
                    TRANSFORM({UNSIGNED4 d},
                              SELF.d :=C2GD(COUNTER)));
  RETURN SET(SORT(ds,d),d) + [GD];
END;
```

- The **GD** definition (Gregorian Date) uses the **Today()** function from the Date standard library to get the current date as an UNSIGNED4 in YYYYMMDD format.
- The **JD** definition (Julian Date) then uses the **FromGregorianCalendar()** function to get the number of days since January 1, 1 AD for that GD date.

- The **dates** function creates a set of the dates for which to generate “sales” records. By producing a set, subsequent code can use the date values by simply indexing into the set.

# Generating Data to Spray(Import)



## GenSalesRec – Generate “sales” dates:

```
SET OF UNSIGNED4 dates := FUNCTION
  GD      := Std.Date.Today();
  JD      := Std.Date.FromGregorianCalendar(GD);
  C2GD(C) := Std.Date.ToGregorianCalendar(JD-C);
  ds      := DATASET(Days-1,
                    TRANSFORM({UNSIGNED4 d},
                              SELF.d :=C2GD(COUNTER)));
  RETURN SET(SORT(ds,d),d) + [GD];
END;
```

- The **C2JD** function takes an integer parameter (**C**) and uses the **ToGregorianCalendar()** function to return YYYYMMDD date values for the days preceding the **GD**.
- The **ds** definition creates a DATASET of the YYYYMMDD date values for all the days preceding the **GD**.
- The **RETURN** expression creates a SET OF UNSIGNED4 to return by using the **SET()** function on the sorted **ds** dataset to create a set of its **d** values, then appending to it the **[GD]** one-element set.

# Generating Data to Spray(Import)



## GenSalesRec – Generate store records for each date:

```
GenRecs := DATASET(RecCnt, TRANSFORM(rec,  
                                     SELF.StoreID    := COUNTER % Stores + 1,  
                                     SELF.TransDate   := dates[COUNTER % Days + 1],  
                                     SELF.Sales       := []));
```

- The **GenRecs** definition generates the number of records defined by the **RecCnt** definition (in this case, 300). This creates the parent records of the nested child dataset that will contain all the generated data.
- It populates each record's **StoreID** field with the “store identifier” number (in this case, 1-100). The **TransDate** field contains the ordinal position of which date to use from the dates set (in this case, the numbers 1-3). The **Sales** field is initialized to an empty set ([]) to be populated later.
- This results in the set of parent records for the nested child dataset, where the **StoreID** and **TransDate** fields constitute a unique key -- each **StoreID** number appears in three records, each with a different **TransDate**.

# Generating Data to Spray(Import)



## GenSalesRec – Generate a “sales” record for each SKU:

```
GenSKUs(UNSIGNED2 ID, UNSIGNED1 dt) :=  
  PROJECT(SKUs,  
    TRANSFORM(SKUrec,  
      Start := FUNCTION //local function in TRANSFORM  
        Cnt      := COUNT(Dates)*SKUcnt;  
        Plus     := (dt*SKUcnt)-SKUcnt;  
        Prelim  := (ID-1) * Cnt;  
        RETURN Prelim + Plus;  
      END;
```

- The **GenSKUs()** function generates the Sales nested child dataset records for one StoreID (passed as the **ID** parameter) on one date (passed as the **dt** parameter). It uses the PROJECT operation on the previously defined SKUs record set to create the record set.
- The **Start** function embedded within the TRANSFORM uses the **ID** and **dt** parameters to calculate the starting point within the Sales dataset for the previously generated “sales” numbers for each specific **StoreID** and date value.



# Generating Data to Spray(Import)



## GenSalesRec – Generate a “sales” record for each SKU:

```
SELF.SKU      := LEFT.SKU;  
C             := SalesCnt(start+COUNTER);  
SELF.NumSold  := C;  
SELF.TotalAmt := SKUprice(LEFT.SKU)*C)); //End TRANSFORM started on last slide
```

- The **SELF.SKU** field gets the **LEFT.SKU** field value from the SKUs dataset, while the **TotalAmt** field defaults to zero.
- The **C** definition value comes from the **SalesCnt()** **DICTIONARY** function, passing it the **ID** record number to get by adding the COUNTER value to the return value from the Start function. This is then used in the **NumSold** expression and the **TotalAmt** field.

# Generating Data to Spray(Import)



## GenSalesRec – Add SKU “sales” to each store for each date:

```
SoldRpt :=  
  PROJECT(SORT(GenRecs,TransDate,StoreID),  
    TRANSFORM(rec,  
      dt      := $.PosInSet(dates,LEFT.TransDate);  
      SELF.Sales := GenSKUs(LEFT.StoreID,dt);  
      SELF      := LEFT));
```

- The **SoldRpt** definition combines the **GenRecs** records with their appropriate **GenSKUs()** record sets to create the Sales nested child dataset. It uses the **PosInSet()** function to get the ordinal position of the **LEFT.TransDate** in the dates set to pass on to the **GenSKUs()** function.

# Generating Data to Spray(Import)



## GenSalesRec – Two returns

```
EXPORT GenSalesRecs := MODULE
  EXPORT Recs := SoldRpt;
  EXPORT Date := Dates;
END;
```

- The **GenSalesRecs** MODULE structure allows us to use both the generated data (Recs) and the set of dates for the data. That way, we can use the data generated to populate the data that we will write for the first two examples. The first will work with XML files, while the second will be JSON files.

# An XML Example



## Defining the XML Files

- Now that we have code to generate data records, we're going to use it to create some XML files to be automatically sprayed and processed using our **AutoSpray** tool. This XML example will generate the XML data files along with the **Semaphore** file to instigate the **AutoSpray** functionality. Once sprayed, we'll "process" the sprayed files by simply adding them to a SuperFile (a common first step in production ETL work).
- We'll start by defining the layout of the files. This code is in the workshop Sales.ecl file

# An XML Example



## Sales.ecl

```
EXPORT Sales := MODULE
  EXPORT SFname := '~RTTEST::IN::SF_XMLfiles';
  EXPORT SKUrec := RECORD
    STRING5      SKU{XPATH('sku')};
    UNSIGNED2     NumSold{XPATH('numsold')};
    UDECIMAL8     TotalAmt{XPATH('totalamt')};
  END;
  EXPORT Layout := RECORD
    UNSIGNED2     StoreID{XPATH('storeid')};
    UNSIGNED4     TransDate{XPATH('transdate')};
    DATASET(SKUrec) Sales{XPATH('/sales')};
  END;
  EXPORT File := DATASET(SFname,Layout,
                        XML('Dataset/Row'));
END;
```

- The **SFname** definition is the SuperFile that will contain all the data files.
- Because these are XML files, the RECORD structures need to make use of the **XPATH()** field modifier to explicitly define the XML tag names to use. That's why we can't just use the structure of the record set generated by the **GenSalesRecs** code, but you will note that the data types and field names are all the same.

# An XML Example



## MOD\_Events.ecl (XML)

```
IMPORT $.^;  
EXPORT Mod_Events := MODULE(ProcessAutomation.Mod_Events)  
  EXPORT Who := 'XML';  
END;
```

- This instance of the **Mod\_Events MODULE** inherits all the EXPORT definitions from the root **Mod\_Events**. It then overrides the **Who** definition and allows all the rest to default to their inherited values.

# An XML Example



## Defining the Interface – I\_XML\_Spray.ecl

```
IMPORT $.^;
EXPORT I_XMLSpray :=
    MODULE(ProcessAutomation.I_AutoSprayFiles)
    EXPORT STRING IP      := IF(__CONTAINERIZED__,
                                'localhost',
                                '10.0.2.15');

    EXPORT STRING LZpath :=
        '/var/lib/HPCCSystems/mydropzone/';
    EXPORT STRING FlagFile      := 'XMLspray.csv';
    EXPORT STRING SprayTargetDir := '~RTTEST::IN::';
    EXPORT STRING NewFileEvent  := $.Mod_Events.Spray;
    EXPORT STRING EventToPush   :=
        $.Mod_Events.Process;
    EXPORT UNSIGNED1 SprayType  := 3;
    EXPORT UNSIGNED1 PauseSecs  := 5;
END;
```

- The **I\_XMLSpray** MODULE provides the concrete instance of the root **I\_AutoSprayFiles INTERFACE** that the XML spray will use. It only overrides those members that are specific to the XML spray and defaults all others.
- The **IP** and **LZpath** definitions must be here, because they were defaulted to empty strings in the root **I\_AutoSprayFiles INTERFACE**.

# An XML Example



## More notes on the Interface

- If you are working in a shop where you only have a single HPCC Systems environment, then you could simply make that environment's IP and LZpath values the defaults, which would eliminate the need to override them here.
- However, if you have more than one HPCC Systems environment to work with, leaving the defaults empty (thus requiring the local overrides) would be the best way to go.
- Of course, you could also default to the most common environment, which would only require overriding for those projects that AutoSpray to any other environment.



# An XML Example



## And even more notes on the Interface...

- The **IP** definition to uses the **\_\_CONTAINERIZED\_\_** compile time constant because in developing this workshop we worked with both Docker (cloud-native) and VirtualBox VM (bare-metal) platforms to ensure it all works with both. This **IF()** function automatically detects whether the platform is running on Docker or VirtualBox, making the IP setting appropriate for each.
- The **FlagFile** and **SprayTargetDir** definitions always need to be overridden because they are always going to be specific to the project.
- The **NewFileEvent** and **EventToPush** definitions are overridden here because they are defaulted to blank in the root **I\_AutoSprayFiles INTERFACE**. These could also be defaulted to the standard **\$.Mod\_Events** entries in the root interface and only overridden when applicable.
- The **SprayType** definition always needs to be overridden because it is always going to be specific to the project. The **PauseSecs** definition is overridden here because I want to allow a five-second pause to make sure the “process” code has time to execute and reset itself to process the next file.

# An XML Example



## The Data File Process Code – BWR\_ProcessXML

```
IMPORT $.^, Std;
filename := EVENTEXTRA('FileName');
SFname := $.Sales.SFname;
isSF := Std.File.SuperFileExists(SFname);
NewSF := IF(~isSF,
            Std.File.CreateSuperFile(SFname));
AddSF := Std.File.AddSuperFile(SFname,filename);

AddFile := NOTHOR(SEQUENTIAL(NewSF,AddSF));

LogText := 'New Subfile Count: ' +
            NOTHOR(Std.File.GetSuperFileSubCount(SFname));
LogEvent :=
OUTPUT(ProcessAutomation.LogOut(LogText),
        NAMED('NotifyRun'),EXTEND);
SEQUENTIAL(AddFile,LogEvent)
: WHEN(EVENT($.Mod_Events.Process,'*'));
```

- The **filename** definition gets its value from the **EVENTEXTRA()** function, which extracts it from the '**FileName**' tag in the XML text passed as the **NOTIFY()** function's second parameter. That **NOTIFY()** function is coming from the **PushEvent()** function in the **Mod\_AutoSprayFiles.ecl** file (discussed previously), which instigates the execution of this workunit to process each file after it has been sprayed.

# An XML Example



## More info on the Data File Process Code:

- The **SFname** definition is a simple re-definition of the superfile name in the **\$.Sales.SFname** definition. The **isSF** definition uses the **Std.File.SuperFileExists()** function to detect whether the superfile has already been created. The **NewSF** definition uses the one-parameter form of the **IF()** function (only valid when the true parameter is an Action) to create the superfile if it doesn't already exist. Then the **AddSF** definition uses the **Std.File.AddSuperFile()** function to add the filename to the list of sub-files in the superfile's metadata.
- The **AddFile** definition uses the **SEQUENTIAL** action to first create the **SFname** superfile (if it doesn't already exist) then adds the filename to the superfile. This is wrapped within a **NOTHOR** action to ensure that it only executes on hThor, because all this is just maintenance to the DFU's metadata and would create an error condition if it were simultaneously executed on each node of your Thor cluster.
- The **LogText** definition creates the text for the log entry, using the **Std.File.GetSuperFileSubCount()** function wrapped in a **NOTHOR** action to get the current count total of subfiles in the superfile. Then, the **LogEvent** definition defines the **OUTPUT** action that writes that log entry.
- The final **SEQUENTIAL** action adds the files and then logs the addition, but only **WHEN** the workunit is triggered for execution by the appearance of the **\$.Mod\_Events.Process** event.

# An XML Example



## Creating the XML Files – BWR\_GenSalesFiles.ecl

- This code emulates the process of FTPing data files to your Landing Zone. It first writes all the XML data files. Then, only when they are all on the Landing Zone, it writes the semaphore file that the AutoSpray is looking for. This order is important: the data files need to be fully present before the semaphore file instigates the AutoSpray process.

# An XML Example



## Start the Code – BWR\_GenSalesFiles.ecl

```
IMPORT Std, $.^;
```

```
CapIt(STRING s) := ProcessAutomation.Mod_Utils.CapCaret(s);  
SoldRpt         := PROJECT(ProcessAutomation.GenSalesRecs.Recs,$.Sales.Layout);  
dates           := ProcessAutomation.GenSalesRecs.Date;
```

- The **CapIt()** function redefines the root **Mod\_Utils.CapCaret()** function purely for cosmetic purposes – I needed to call it several times where it would have caused code formatting problems in this presentation.
- The **SoldRpt** definition uses PROJECT to reformat the data records to the **\$.Sales.Layout RECORD** structures so the XPATH modifiers would be available to correctly build the XML files.
- The **dates** definition retrieves the set of dates used for the data generation.

# An XML Example



## Landing Zone Filenames – BWR\_GenSalesFiles.ecl

```
IP := $.I_XMLSpray.IP;
LZpath := '~file::' + IP + '::var::lib::' + 'HPCCSystems::mydropzone::';
LZfileName(String s,n) := s + '_Sales_' + dates[n] + '.xml';
file1(String s) := LZpath + LZfileName(s,1);
file2(String s) := LZpath + LZfileName(s,2);
file3(String s) := LZpath + LZfileName(s,3);
```

- The **LZpath** definition is the path to your Landing Zone. To write or read files from the Landing Zone, the filename must begin with **~file::** followed by the IP of the Landing Zone (for my local Docker environment, that is localhost, but for a VirtualBox VM it is the second inet IP you see by running the **ip addr show** command), then the full path to the directory to contain the file (found by looking at the top of the Landing Zone page in ECL Watch in the square brackets attached to the mydropzone folder), replacing the directory delimiters with double colons (::).
- The **LZfileName()** function builds the individual file names to write to disk by concatenating the filename's prefix (passed as the s parameter) to the **'\_Sales\_'** constant. Then adding the dates set element specified by the **n** parameter, and finishing off with the **'.xml'** constant.
- The **file1()** function concatenates the **LZpath** to the **LZfileName()** to create the fully qualified path to the first file to write to disk. The **file2()** and **file3()** functions do the same for the second and third files.

# An XML Example



## Emulate FTPing the Data Files – BWR\_GenSalesFiles.ecl

```
FTPfiles := PARALLEL(  
  OUTPUT(SoldRpt[ 1..100],,CapIt(file1('XML')),XML,OVERWRITE),  
  OUTPUT(SoldRpt[101..200],,CapIt(file2('XML')),XML,OVERWRITE),  
  OUTPUT(SoldRpt[201..300],,CapIt(file3('XML')),XML,OVERWRITE));
```

- The **FTPfiles** action uses **PARALLEL** to allow all three **OUTPUT** actions to execute in parallel.
- NOTE: **PARALLEL** does not force parallel execution, it only tells the compiler that parallel execution is okay to do, if the compiler chooses to.
- Each **OUTPUT** action writes 100 records -- the first will all be from the first date, and so on.

# An XML Example



## Create the semaphore file – BWR\_GenSalesFiles.ecl

```
SemRec := {STRING s, STRING n, STRING t};  
fname(STRING s) := ProcessAutomation.Mod_Utils.RawFilename(s);  
XMLfiles := DATASET([  
    {fname(file1('XML')), '65535', 'Row'},  
    {fname(file2('XML')), '65535', 'Row'},  
    {fname(file3('XML')), '65535', 'Row'}], SemRec);
```

- The **XMLfiles** definition is a simple inline **DATASET** that creates the set of records for the semaphore file. The file does not need to contain the path to the files (because the semaphore file will be placed in the same directory on the landing Zone as the data files), so the **fname()** function uses the **Mod\_Utils.RawFilename()** function to extract just the filenames.



# An XML Example



## Emulate FTPing the semaphore file – BWR\_GenSalesFiles.ecl

```
FTPsemaphore := OUTPUT(XMLfiles,,CapIt(LZpath + 'XMLspray.csv'),CSV,OVERWRITE);
```

- The **FTPsemaphore** definition is the **OUTPUT** action that writes the semaphore file to the Landing Zone, emulating the process of FTPing a file to the Landing Zone.

## Write data files first, then the semaphore:

```
SEQUENTIAL(FTPfiles,FTPsemaphore);
```

- The **SEQUENTIAL** action ensures that the data files are fully written before the semaphore file.

# An XML Example



## Launching the Workunits! (Finally)

```
First -- BWR_ProcessXML  
Next  -- BWR_SprayXML
```

- You should launch the workunits in reverse order to their execution. That way, the last one to launch is the “first domino” that begins causing all the rest to fall in order.
- Once launched, you can open the monitor DFU workunit launched by BWR\_SprayXML and edit the Job Name field so that you may have multiple file monitors running that can be easily distinguished in the DFU workunits list.

# An XML Example



## Instigating the AutoSpray – Uploading the demo files


- Once the workunits are all running and waiting for their respective events to trigger the actual work, only then are you ready to actually place files on your Landing Zone and have them automatically sprayed to your environment.
- For this example, the data files to spray are generated by the running the **BWR\_GenSalesFiles.ecl** code, previously discussed.
- **Note:** Normally, you would simply upload the files to your Landing Zone in your usual manner. For my testing purposes, I just use the Upload button on the ECL Watch Landing Zones page, but for production purposes you would more likely be using an FTP solution to place the files there.

# An XML Example



## Watch it all work!

- After all the files have been uploaded, the semaphore file's presence on the Landing Zone will trigger the process to start. You can look at the DFU workunit's workunit identifier to see what time it launched and predict about when the process will trigger. The file monitor checks for the semaphore file every 15 minutes. Once it has run, the ECL Watch Logical Files page will look something like this:

Logical Name	Owner	Super Owner
 <a href="#">rttest::in:sf_xmlfiles</a>	Bob	
 <a href="#">rttest::in:xml_sales_20240627.xml</a>	Bob	rttest::in:sf_xmlfiles
 <a href="#">rttest::in:xml_sales_20240626.xml</a>	Bob	rttest::in:sf_xmlfiles
 <a href="#">rttest::in:xml_sales_20240625.xml</a>	Bob	rttest::in:sf_xmlfiles

As you see:

1. The three datasets were sprayed automatically
2. The superfile was automatically created
3. Each dataset was automatically added to the superfile



## Implementing the AutoSpray Tool with JSON Demo

# A JSON Example



## Defining the files:

- Now that we have automatically sprayed XML files, we'll do the same with JSON files (which requires some subtle differences). Because this example is very similar to the previous XML example, I will only discuss the differences between the two sets of code.

# A JSON Example



## Defining the files (Sales.ecl):

```
EXPORT Sales := MODULE
  EXPORT SFname :=
    '~RTTEST::IN::SF_JSONfiles';
  EXPORT SKUrec := RECORD
    STRING5      SKU;
    UNSIGNED2    NumSold;
    UDECIMAL8    TotalAmt;
  END;
  EXPORT Layout := RECORD
    UNSIGNED2    StoreID;
    UNSIGNED4    TransDate;
    DATASET(SKUrec) Sales{XPATH('sales/')};
  END;
  EXPORT File := DATASET(SFname,Layout,
                        JSON('Row/'));
END;
```

- The first difference here is in the **XPATH()** attributes. The XML version of this code explicitly named the XML tags for every field, but this JSON version only specifies the Sales name as “sales/” which indicates the name of the JSON array of name/value pairs (the JSON version of a nested child dataset).
- The only other difference is the **JSON('Row/')** attribute of the DATASET definition. The XML version started with a slash ( /Row ) while the JSON version must end with one ( Row/ ).

# A JSON Example



## Defining the events (Mod\_Events.ecl):

```
IMPORT $.^;  
EXPORT Mod_Events := MODULE(ProcessAutomation.Mod_Events)  
  EXPORT Who := 'JSON';  
END;
```

- The only difference is the replacement of XML with JSON in the **Who** definition.



# A JSON Example



## Defining the interface (I\_JSONSpray.ecl):

```
IMPORT $.^;
EXPORT I_XMLSpray := MODULE(ProcessAutomation.I_AutoSprayFiles)
  EXPORT STRING IP := IF(__CONTAINERIZED__, 'localhost', '10.0.2.15');
  EXPORT STRING LZpath      := '/var/lib/HPCCSystems/mydropzone/';
  EXPORT STRING FlagFile    := 'JSONspray.csv';
  EXPORT STRING SprayTargetDir := '~RTTEST::IN::';
  EXPORT STRING NewFileEvent := $.Mod_Events.Spray;
  EXPORT STRING EventToPush  := $.Mod_Events.Process;
  EXPORT STRING EventToLaunch := $.Mod_Events.Final;
  EXPORT UNSIGNED1 SprayType := 4;
  EXPORT UNSIGNED1 PauseSecs := 5;
END;
```

- The differences here are the **FlagFile** name and the **SprayType** definition value has been changed to 4 to indicate JSON files. The **EventToLaunch** has been added to specify that there will be a “final” event to push once all files have been sprayed.

# A JSON Example



## The data file Process code (BWR\_ProcessJSON.ecl):

- The "process" work is in this file.
- There are no differences in this code from the XML version, they both just add the newly-sprayed file to the superfile.

## The data file Spray(Import) code (BWR\_SprayJSON.ecl):

```
IMPORT $.^;
```

```
ProcessAutomation.Mod_AutoSprayFiles($.I_JSONSpray).SprayFiles :  
WHEN(EVENT($.I_JSONSpray.NewFileEvent, '*'));  
ProcessAutomation.Mod_AutoSprayFiles($.I_JSONSpray).WaitForFile;
```

- The only difference is the replacement of *XML* with *JSON* in each line.

# A JSON Example



## Creating the JSON files (BWR\_GenSalesFiles.ecl):

```
//Partial code
LZfileName(STRING s,n) := s + '_Sales_' + dates[n] + '.json';
//these OUTPUTs emulate FTPing the data files
FTPfiles := PARALLEL(
  OUTPUT(SoldRpt[1..100],,CapIt(file1('JSON')),JSON,OVERWRITE),
  OUTPUT(SoldRpt[101..200],,CapIt(file2('JSON')),JSON,OVERWRITE),
  OUTPUT(SoldRpt[201..300],,CapIt(file3('JSON')),JSON,OVERWRITE));
JSONfiles := DATASET([
  {fname(file1('JSON')), '65535', 'Row/'},
  {fname(file2('JSON')), '65535', 'Row/'},
  {fname(file3('JSON')), '65535', 'Row/'}],SemRec);
```

- The only differences here are:
  1. The **LZfilename()** function expression ends with '.json' instead of '.xml'.
  2. The replacement of every *XML* instance with *JSON*.
  3. The addition of the slash ( / ) to the '**Row/**' text.

# A JSON Example



## Launching the workunits:

- For this JSON case, that order is:
  1. First -- BWR\_GenStatsJSON
  2. Second -- BWR\_ProcessJSON
  3. last -- BWR\_SprayJSON
- With the addition of the **I\_JSONSpray.EventToLaunch** override to specify a final event to launch a process that runs after all files have been sprayed, you need to re-write the code that was in the **BWR\_GenStatsXML.ecl** file to be event-driven and produce a single “report” result instead of four separate values to look at.

# A JSON Example



## Final Processing (BWR\_GenStatsJSON):

```
//Partial code
```

```
Report := DATASET([{SalesTotals, SORT(StoreTbl1, StoreID), SORT(StoreTbl2, StoreID, TransDate)}], RptRec);
```

```
outfile := 'JSON_SalesReport_' + Std.Date.Today() + '_' + Std.Date.CurrentTime();
```

```
OUTPUT(Report,,outfile) : WHEN(EVENT($.Mod_Events.Final,'*'));
```

- This code generates the same stats that the four **OUTPUT()** actions in the *BWR\_GenStatsXML.ecl* file generated. However, this produces a one-record **DATASET** with one **STRING** field containing the two **SalesTotals** values, and two nested child datasets (the **SalesPerStore** and **SalesPerStoreByDate** tables).
- The WHEN() workflow service on the OUTPUT action triggers from the **\$.Mod\_Events.Final** event, which is automatically launched after all the files have been sprayed.

