Technical Paper

Extending SAS® Forecast Server Client

How To Write Plug-Ins

Release Information Content Version: 1.0 June 2016.

Contents

SAS Forecast Server Client 1
Using Lua as the Interface between SAS Forecast Server Client and the Plug-Ins 1
How to Write a Plug-In for SAS Forecast Server Client 2
The info.lua File 2
The run.lua File 2
Input Table and Allocation of Output Librefs 4
Segmentation Strategy Plug-Ins 5
Requirements 5
Example of a Segmentation Strategy Plug-In 5
Modeling Strategy Plug-Ins 8
Requirements 8
Example of a Modeling Strategy Plug-In 9
Putting Everything Together 13
Conclusion 17
Resources 17

SAS Forecast Server Client

SAS Forecast Server Client is a new web-based interface that is shipped with SAS® Forecast Server. It promotes modernization of the user experience and reflects SAS' initiative to provide modern, web-based clients that allow anywhere/anytime access. It also promotes product consolidation, integrating capabilities from SAS® Forecast Studio and SAS® Time Series Studio. SAS Forecast Server Client offers an improved interface that provides the following benefits:

- segmentation, clustering, and volume grouping
- access to additional modeling techniques
- a modern, web-based interface that eliminates the need to maintain local client installations
- a best-practice approach to large-scale forecasting
- out-of-the-box functionality to support a majority of cases
- flexibility and extensibility to fit any scenario

SAS Forecast Server Client is shipped with two built-in strategies that will satisfy about 80% of common forecasting use cases: a rule-based segmentation strategy and a hierarchical forecasting modeling strategy. This product is also shipped with two plug-ins that illustrate its flexibility and extensibility: a code-based segmentation strategy (SAS® Demand Classification and Clustering) and a code-based modeling strategy (SAS Multistage Modeling).

You can think of a plug-in as a macro with a list of arguments that control how the macro functions on data or instructions. This architecture enables you to define customized segmentation and forecasting methods while you maintain and manage your projects in SAS Forecast Server Client. Similar to the built-in strategies, the plug-ins can be imported and registered as building blocks. Furthermore, because the plug-ins are registered strategies, they can be reused by other users who want to apply them to their own projects.

Using Lua as the Interface between SAS Forecast Server Client and the Plug-Ins

Lua is a lightweight programming language that is designed as a scripting language similar to popular scripting languages such as Perl and Python but is much easier to learn. Lua is a dynamically entered language that is intended for use as an extension or scripting language. It supports only a small number of atomic data structures such as Boolean values, numbers, and strings. Typical data structures such as arrays, sets, lists, and records can be represented using Lua's single native data structure, the table, which is essentially an associative array.

The LUA procedure, available in the third maintenance release of SAS® 9.4 and later, is a standard Lua interface between SAS Forecast Server Client and the plug-ins. In addition, you can use SAS extensions to Lua—in particular, you can use the **sas.submit** function in order to submit SAS code within Lua. For more information about programming in Lua, see https://www.lua.org/pil/contents.html.

How to Write a Plug-In for SAS Forecast Server Client

To write a plug-in for SAS Forecast Server Client, you need to create two files: info.lua and run.lua. The info.lua file contains the meta-information about the plug-in, and the run.lua file contains the run-time code that SAS Forecast Server Client executes within a project run.

The info.lua File

The info.lua file contains the following types of information:

- name, type, and description
- name of the plug-in run-time code
- settings and their corresponding default values
- all possible segmentation values
- optional segments and the corresponding optional modeling strategy map to each possible segment value

For each of the settings, you need to provide the name, description, default value, and level to be displayed in the GUI. The value for each setting is of the character type. You need to convert the character setting values to numeric (using the **tonumber()** Lua function) in the run-time code. The level can be either basic or advanced. By default, the GUI hides the advanced settings until you explicitly request that they be displayed. You can use this default behavior to hide advanced settings from general users.

The following code shows an example of an info.lua file:

```
return {
   FswbSdkVersion=14.1, --indicate the required version number of the SAS Forecast
Server Client
   name='My code sample', --name of the plug-in
   description='This is my code sample', --description of the plug-in
   type='segmentation',  --plug-in type. It can either be segmentation or forecast
   main='run.lua',  --name of the run-time code
   version=1.0,  --version number of the plug-in code
   settings= {{name='foo', description='This is foo', defaultValue='5', level=
'basic'},
             {name='baz', description='This is baz', defaultValue='BAZ', level =
'advanced'}},
             --list of plug-in settings
   segments= { 'A', 'B', 'C'}  --indicate the possible segments generated by the
plug-in
}
```

The run.lua File

The run.lua file defines the run-time code that will be submitted in a project run process. During the run time, SAS Forecast Server Client reads the settings from the info.lua file and updates them with project-specific values that you specify. SAS Forecast Server Client also collects the project and node information (data source, dependent variables, and so on) to form the following two Lua objects: **_G.project** and **_G.node**.

The **_G.project** Lua object contains the basic information about the project. You need only the information from the **dataSpec** subobject for the plug-ins. The structure of the **_G.project** Lua object is as follows:

```
project
    - dataSpec
        - dependent_variable
            - hierarchy_aggregation
            - name
            - horizon_start
            - time_interval_aggregation
            - zero_interpretation
            - missing_trim
            - missing_interpretation
        - byvars
            - <byvar1>
            ...
        - time_id
            - name
            - interval_name
            - seasonality
            - default_format
            - start
            - end
            - shift
            - weekday
            - multiplier
            - max_shift
            - datetime
            - weekend_days
            - custom_format
        -independent_variables
            -<x1>
                - missing_interpretation
                - name
                - input_usage
                - time_interval_aggregation
                - zero_interpretation
                - missing_trim
                - hierarchy_aggregation
            ...
        - lead
        - reconciliation_level
        - confidence_limit
```

You can follow this structure to retrieve the values of the settings. For example, you can retrieve the name of the dependent variable by using **_G.project.dataSpec.dependent_variable.name**, the value of the forecast lead by using **_G.project.dataSpec.lead**, and the number of BY variables by using **#_G.project.dataSpec.byvars**.

The **_G.node** Lua object contains the node identification and user settings that are gathered from the GUI. The following code shows the structure of the **_G.node** Lua object:

```
node
    - id
    - settings
        - <setting1>
        ...
```

For example, you can retrieve the value of the node identification by using **_G.node.id** and the value of a user setting xyz by using **_G.node.settings.xyz**.

Input Table and Allocation of Output Librefs

SAS Forecast Server Client provides built-in librefs for accessing the input tables for segmentation and modeling strategy plug-ins. You can use the **allocate_product_library** function of the **_G.project** Lua object—**_G.project:allocate_product_library(<libname>, <node id>, <level>)**—to assign librefs that point to the output locations.

For segmentation strategy plug-ins:

- Input data: _IMPORTS.DATA
- Output:
  - Assign libref **_G.project:allocate_product_library("_OUTLIB", _G.node.id, lowestLevel)**
  - Output data: _OUTLIB.DATA

For modeling strategy plug-ins:

- Input data: _IMPORTS.SEGMENT
- Output:
  - Assign libref **_G.project:allocate_product_library("_OUTLIB", _G.node.id, level)** for a particular hierarchy level
  - Output data:
    - _OUTLIB.OUTFOR
    - _OUTLIB.OUTSTAT
    - _OUTLIB.OUTSUM
    - _OUTLIB.MODELINFO
    - _OUTLIB.RECFOR
    - _OUTLIB.RECSTAT

You should store the plug-in code in the following location:

```
<SAS Install>/Lev1/AppData/SASForecastServerClient/source/custom/plugins
```

You need to create a subdirectory for each of the plug-ins and save the info.lua and run.lua files to the corresponding subdirectory. If you have any module files, you can save them directly to the subdirectory or create a module subdirectory and save the module files there.

The following code shows an example of the directory structure:

```
/plugins/<plugin_1>
/plugins/<plugin_1>/info.lua
/plugins/<plugin_1>/run.lua

/plugins/<plugin_1>/<module>.lua
/plugins/<plugin_1>/<module>
/plugins/<plugin_1>/<module>/<module>.lua
```

Segmentation Strategy Plug-Ins

Requirements

The segmentation strategy plug-ins enable you to use all the analytics that SAS offers (such as the procedures in SAS/STAT®, SAS/ETS®, and SAS® Enterprise Miner™) to segment the data so that proper modeling strategies can be applied to each segment. Note that SAS/STAT and SAS/ETS are bundled with SAS Forecast Server.

The input table for any segmentation strategy is _IMPORTS.DATA.

The _IMPORTS libref is a predefined system library that points to the location where the data set Data contains the input data that are specified in the INPUT node.

The output table of the segmentation strategy is _OUTLIB.DATA.

The Data data set should contain all the variables in the input data plus an additional variable called DC_BY. This additional variable is used to separate the input data into segments that are to be consumed by the models that are applied to each segment. You can use the **allocate_product_library _G.project** Lua object function to allocate the librefs for the lowest level in the hierarchy. For example, you can use **_G.project:allocate_product_library("_OUTLIB", _G.node.id, lowestLevel)**, where **lowestLevel** is a numeric value that represents the numbering of the lowest level in the hierarchy (that is, the number of BY variables).

Example of a Segmentation Strategy Plug-In

The following example illustrates a segmentation strategy plug-in that splits the data into four segments, based on either systematic sampling or random sampling. First you need to create the info.lua file. The **type** must be set to “segmentation” so that SAS Forecast Server Client recognizes the plug-in as a segmentation strategy type. The **main** argument is the name of the run-time code (usually named run.lua). Two settings are specified in the file: **segMethod** and **randomSeed**. The default value of **segMethod** is SYSTEMATIC and the level is basic. This means that **segMethod** is shown to the user and the value SYSTEMATIC is displaced by default. The **randomSeed** setting is used for random sampling, and it also has a default value. Because the level is advanced, users do not see the setting by default. However, users can request to see all settings and change the values if they want. The **segments** argument contains the names of the four possible resulting segments. Note that five segments are shown in the process flow diagram because there is always a segment called “unsegmented” (also called the catch-all segment). This segment is used to capture any series that cannot be put into any of the other segments.

```
-- info.lua for sg_5 plug-in
return {
   FswbSdkVersion=14.1,
   name='Code-Based Segmentation with Five Segments',
   description='Code-Based Segmentation with Five Segments',
   type='segmentation',
   main='run.lua',
   version=1.0,
   settings= {
     {
       name = 'segMethod',
       description = 'Segmentation method. Valid values are SYSTEMATIC and RANDOM',
       defaultValue = 'SYSTEMATIC',
       level="basic"
     },
```

```
      {
        name = 'randomSeed',
        description = 'Random seed value for random segmentation',
        defaultValue = '12345',
        level="advanced"
      }
    },  -- end bracket for the settings table
    segments= {'Segment_1',
               'Segment_2',
               'Segment_3',
               'Segment_4'}
} -- end of return
```

If you have a registered modeling strategy that you want SAS Forecast Server Client to map to when it generates the process flow, you can change the segments table to include this strategy. For example, the following strategy assigns the Segment_1 series to use modeling strategy ‘MS1’ and assigns the Segment_2 series to use modeling strategy ‘MS2’:

```
segments= {
            {value = 'Segment_1', strategy = 'MS1'},
            {value = 'Segment_2', strategy = 'MS2'},
            {value = 'Segment_3'},
            {value = 'Segment_4'}
          }
```

For segments that are not mapped to a modeling strategy, the default modeling strategy (unsegmented strategy) is considered when the process flow diagram is built.

Next you need to create the run-time code (run.lua) for the plug-in, as shown in the following code example. You start by populating the **args** table that will be used to call the Lua modules. Data specification information (such as dependent variable, time ID, and BY variables) can be accessed via the **_G.project.dataSpec** Lua object. The input data are _IMPORTS.DATA, and the output table will be _OUTLIB.DATA, where the _OUTLIB libref is defined using the **allocate_product_library** function in the **_G.project** object. The user settings can be accessed from the **_G.node.settings** object.

```
-- run.lua for sg_5 plug-in
print("Starting SG_5 plug-in...")

local args = {}

local by = _G.project.dataSpec.byvars;
local bottomLevel = #by;

-- Define the segmentation output libref
_G.project:allocate_product_library("_OUTLIB", _G.node.id, bottomLevel)

--  get I/O data
args.indata  = "_IMPORTS.data"
args.outdata = "_OUTLIB.data"

-- get the user settings
local userSettings = _G.node.settings

-- get byVars info
local byVars
local byVarsComma
```

```
byVars = ''
byVarsComma = ''
for i, v in ipairs(_G.project.dataSpec.byvars) do
  if i == 1 then
    byVars      = _G.project.dataSpec.byvars[i]
    byVarsComma = _G.project.dataSpec.byvars[i]
  else
    byVars      = byVars..' '.._G.project.dataSpec.byvars[i]
    byVarsComma = byVarsComma..' ,'.._G.project.dataSpec.byvars[i]
  end
end
args.byVars      = byVars
args.byVarsComma = byVarsComma


-- submit SAS code with the parameters in the args table
local rc
rc = sas.submit([[
        proc sql;
           create table _distinct_byvar as
           select distinct @byVarsComma@
           from @indata@
           order by @byVarsComma@
           ;
        quit;
        ]], args)

-- check the segmentation method from the user settings
if userSettings.segMethod:upper() == 'SYSTEMATIC' then
  rc = sas.submit([[
          Data _segment_assignment;
             set _distinct_byvar;
             format dc_by $char20.;
             if mod(_n_,5) = 1 then dc_by = 'Segment_1';
             else if mod(_n_,5) = 2 then dc_by = 'Segment_2';
             else if mod(_n_,5) = 3 then dc_by = 'Segment_3';
             else if mod(_n_,5) = 4 then dc_by = 'Segment_4';
             else if mod(_n_,5) = 0 then dc_by = 'NA';
          run;
          ]], args)
else
  -- use random assignment
  args.randomSeed = userSettings.randomSeed
  rc = sas.submit([[
          Data _segment_assignment;
             set _distinct_byvar;
             format dc_by $char20.;
             randomNumber = ranuni(@randomSeed@);
             if randomNumber <= 0.2 then dc_by = 'Segment_1';
             else if randomNumber <= 0.4 then dc_by = 'Segment_2';
             else if randomNumber <= 0.6 then dc_by = 'Segment_3';
             else if randomNumber <= 0.8 then dc_by = 'Segment_4';
             else dc_by = 'NA';
             drop randomNumber;
          run;
          ]], args)
end

rc = sas.submit([[
        proc sort data = @indata@ out = _indata_tmp;
           by @byVars@;
        run;
```

```
        data @outdata@;
           merge _indata_tmp _segment_assignment;
           by @byVars@;
        run;
        ]], args)

print("SG_5 plug-in has finished.")
```

Modeling Strategy Plug-Ins

Requirements

The modeling strategy plug-ins enable you to use all the analytics that SAS offers (such as the procedures in SAS/STAT, SAS/ETS, and SAS Enterprise Miner) to model the data and generate forecasts. Note that SAS/STAT and SAS/ETS are bundled with SAS Forecast Server.

The input table for any modeling strategies is _IMPORTS.SEGMENT.

Similar to the segmentation node, the _IMPORTS libref points to the location where the data set Segment contains the subset of the input data for a particular segment.

The outputs of the modeling strategy plug-in need to conform to the standard forecast outputs from SAS Forecast Server in order for SAS Forecast Server Client to recognize the forecasts that the plug-ins generate.

Again, you can use the **allocate_product_library** project function to allocate the libref for each level in the hierarchy, as shown by the following examples:

_G.project:allocate_product_library("_OUTL1", _G.node.id, 1)

_G.project:allocate_product_library("_OUTL2", _G.node.id, 2)

_G.project:allocate_product_library("_OUTL3", _G.node.id, 3)

The following standard SAS Forecast Server output tables are required:

- OUTFOR (original forecasts from the forecast models)
- OUTSTAT (forecast accuracy measures such as MAPE and RMSE)
- OUTMODELINFO (types of models considered for each series)
- RECFOR (reconciled forecasts)
- RECSTAT (reconciled forecast accuracy measures such as MAPE and RMSE)

It is not easy to handcraft the required output table with the required variables in it. However, you can use the HPFENGINE procedure to run through the forecasts that are generated by whatever approach you consider and generate all the required tables in the correct format. To make this trick work, you must at a minimum prepare the forecast data with all the BY variables, the TimeID variable, the variable that you are forecasting, and the forecasted values of the actual and future variables.

In the following code, a GLM procedure model is used to make predictions for the variable Sale in the PriceData data set. Because the TimeID variable is irrelevant for the PROC GLM model, it is not included in the MODEL statement. The results are output to a table called GLMFOR. The forecasted sale variable is called Sale_Forecast.

```
/*using PROC GLM to fit sale against price*/
proc glm data = sashelp.pricedata noprint;
   by regionName productLine productName;
   model sale = price;
   output out = glmfor p = sale_forecast;
run;
```

You now use the following statements to pass the GLMFOR table to PROC HPFENGINE to generate all the required tables. Basically you ask PROC HPFENGINE to treat the GLMFOR table as external forecasts (see the **exmselect** and **external** keywords in the example). For more information, see *SAS Forecast Server: User's Guide*.

```
/*call PROC HPFENGINE against the GLM model output to get
  the standard forecast outputs */
proc hpfengine data=glmfor out=_NULL_
   outfor = outfor outstat = outstat outsum = outsum outmodelinfo = outmodelinfo
   errorcontrol=(severity=(none) stage=all) lead = 0
   globalselection=exmselect;
   by regionName productLine productName;
   id date interval=MONTH format=MONYY. acc=total notsorted;
   forecast sale;
   external sale_forecast;
run;
```

Example of a Modeling Strategy Plug-In

The following code illustrates a simple modeling strategy that calls the HPF procedure to generate the forecasts and then uses the HPFENGINE procedure to generate the required output tables. The calls to PROC HPF and PROC HPFENGINE are defined in a Lua module that will be loaded in the run-time code. Again you need to first create the info.lua file. The **type** must be set to "forecast" so that SAS Forecast Server Client can recognize the plug-in as a modeling strategy plug-in. The **main** argument is the name of the run-time code (usually called run.lua). Two settings are specified in this example: **seasonality** and **model**. The default value of **seasonality** is 12 and the level is basic, which means that **seasonality** will be shown to the user and the value 12 will be displaced by default. The **model** setting is used to define the type of models for PROC HPF. Because the level is advanced, users will not see the setting by default. However, users can request to see all settings and change the values if they want.

```
-- info.lua for the simple modeling strategy plug-in
return {
  FswbSdkVersion=14.1,
  name="Simple Modeling Plug-In",
  description="Simple Modeling Plug-In",
  type="forecast",
  main="run.lua",
  version=1.0,
  settings= {
    {
      name = 'seasonality',
      description = 'SEASONALITY for TIME ID',
      defaultValue = '12',
```

```
      level = 'basic'
    },
    {
      name = 'model',
      description = 'Model type for ESM models',
      defaultValue = 'BEST',
      level = 'advanced'
    }
  } -- end of settings
} -- end of return
```

Next you need to create a module that returns the functions that are required in the run-time code. It is a good practice to declare the functions in the module as local and to return the local functions via the Lua **return** statement. Only the functions that you specify in the **return** statement will be visible to the run-time code. In the following code, the local function is called **run_hpf_local**, and it is returned as **run_hpf** via the **return** statement. The run-time code can access the function by first loading the module and then calling the **run_hpf** function.

```
-- module to call PROC HPF
local function run_hpf_local(args)
  local rc
  rc = sas.submit([[
    *sort data;
    proc sort data    = @inData@
            out     =  _sorted_data;
       @sortStatement@;
    run;
    *run PROC HPF;
    proc hpf data    = _sorted_data
            out     = _null_
            outfor  = _outFor
            @seasonality@
            lead = @lead@
            ;
       id @idVar@ interval = @idInterval@ acc=total;
       forecast @depVar@ @forecastOptions@;
       @byStatement@;
    run;

    /* run PROC HPFENGINE to generate the required output tables.
      Note that PROC HPF generates all required tables but this is
      for illustration */
    proc hpfengine data           = _outfor(rename=(ACTUAL = @depVar@))
                out           = _null_
                outfor         = @outFor@
                outsum          = @outSum@
                outstat         = @outStat@
                outmodelinfo    = @outModelInfo@
                errorcontrol    = (severity=(none) stage=all)
                globalselection = exmselect
                @seasonality@
                lead = @lead@
                ;
       id @idVar@ interval = @idInterval@ acc=total notsorted;
       forecast @depVar@;
       @byStatement@;
       external predict std lower upper;
    run;

    /* copy data to conform with the output contact requirements.
```

```
    Note that no reconciliation is performed */
  data @recFor@;
    set @outFor@;
    format _RECONSTATUS_ best12.;
    _RECONSTATUS_=0;
  run;
  data @recStat@;
    set @outStat@;
  run;
  ]]
  , args)
 return rc

end

return {run_hpf = run_hpf_local}
```

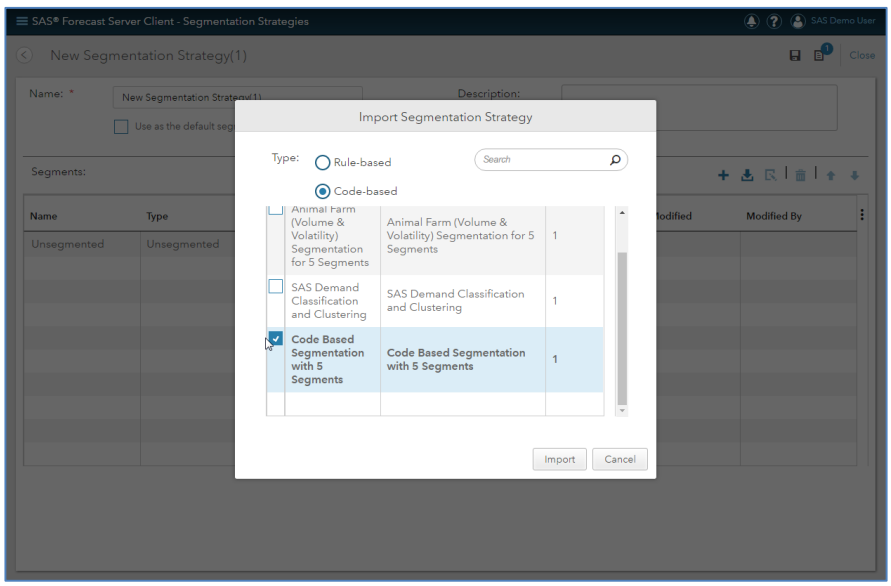
Next you need to create the run-time code (run.lua) for the plug-in. You start by populating the **args** table that will be used to call the Lua modules. Data specification information (such as dependent variable, time ID, and BY variables) can be accessed via the **_G.project.dataSpec** Lua object. The input data table is _IMPORTS.DATA, and the output table is _OUTLIB.DATA, where _OUTLIB librefs are defined using the **allocate_product_library** function in the **_G.project** object. The user settings can be accessed from the **_G.node.settings** object. You need to load the Lua module that contains the **run_hpf** function in the run-time code in order to access the function.

```
-- run.lua for the simple modeling strategy plug-in
print("Starting Simple Modeling Plug-In...")

--[[loading Lua Module containing the function to run PROC HPF.
    The module file is stored in the file system as
source/custom/plugins/ms_1/private_module/run_hpf.lua]]

package.loaded["private_module.run_hpf"] = nil
local ms = require('private_module.run_hpf')

-- prep args, getting values from macro variables
local args = {}

-- define the input data based on the contract
args.inData  = '_IMPORTS.segment'

-- get project info
args.depVar     = _G.project.dataSpec.dependent_variable.name
args.idVar      = _G.project.dataSpec.time_id.name
args.idInterval = _G.project.dataSpec.time_id.interval_name
args.lead       = _G.project.dataSpec.lead

-- load user settings from settings
local userSettings = _G.node.settings

-- check the seasonality spec value. The default seasonality of the
-- time ID will be used if it is not provided
local seasonality = userSettings.seasonality
if seasonality ~= "" then
  args.seasonality = "seasonality = "..seasonality
else
  args.seasonality = ""
end
```

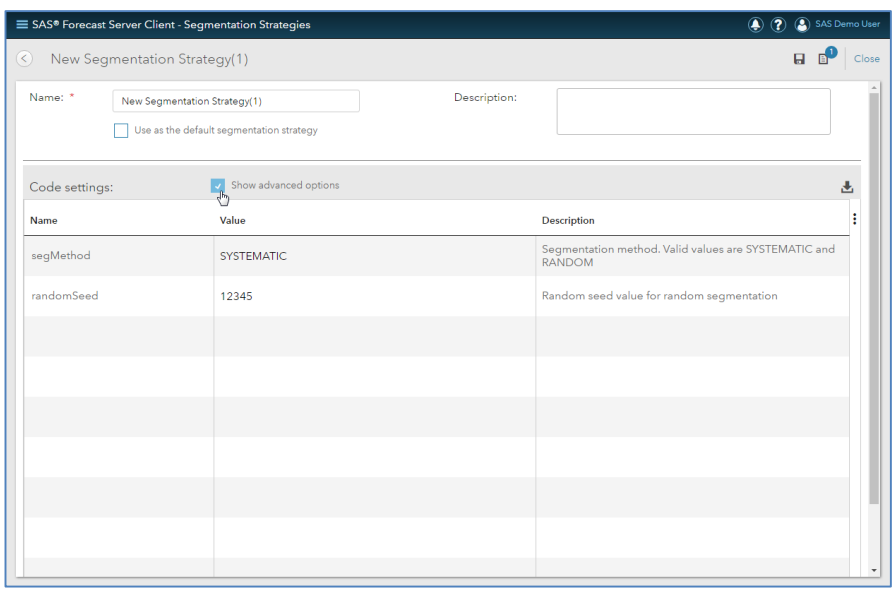
```
-- check the model spec value
-- The default model of PROC HPF will be used if it is not provided
local forecastOptions = userSettings.model
if forecastOptions ~= "" then
  args.forecastOptions = "/model = "..forecastOptions
else
  args.forecastOptions = ""
end

--[[process byVars for each hierarchy level (including the TOP level),
    assign output librefs, and call run_hpf function]]

for i = 0, #_G.project.dataSpec.byvars do

  local byVars
  local sortStatement

  -- initialize byVars
  byVars = ''

  -- no byVars for the TOP level
  if i == 0 then
    args.byStatement = ""
    args.sortStatement = 'by '..args.idVar
  else
    byVars = byVars..' '.._G.project.dataSpec.byvars[i]
    args.byStatement = "by "..byVars
    args.sortStatement = args.byStatement..' '..args.idVar
  end

  -- Define the modeling output librefs
  _G.project:allocate_product_library("_OUTLIB", _G.node.id, i)
  args.outFor        = '_OUTLIB.'..'OUTFOR'
  args.outStat       = '_OUTLIB.'..'OUTSTAT'
  args.outSum        = '_OUTLIB.'..'OUTSUM'
  args.outModelInfo  = '_OUTLIB.'..'OUTMODELINFO'
  args.recFor        = '_OUTLIB.'..'RECFOR'
  args.recStat       = '_OUTLIB.'..'RECSTAT'

  -- run hpf with the args
  local rc
  rc = ms.run_hpf(args)
end

print("Simple Modeling Plug-In has finished.")
```

Putting Everything Together

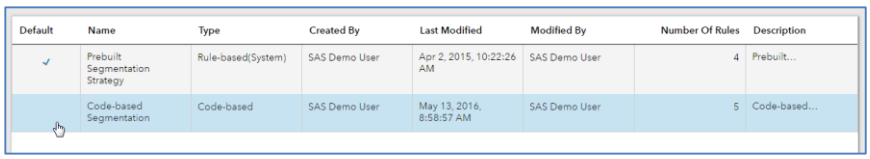
After you save the plug-in codes to the proper location, SAS Forecast Server Client should start picking up the plug-ins without restarting the server. In order to use the plug-ins, you first need to register the plug-ins by creating either a segmentation strategy or a modeling strategy. For example, you can create a new segmentation strategy based on the example plug-in in this document as follows:



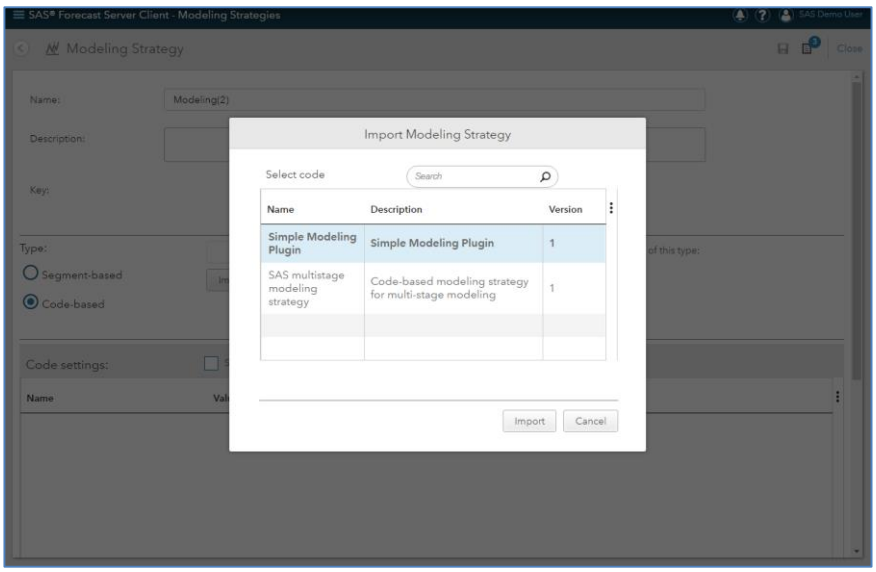
You can select **Show advanced options** to see any advanced setting provided by the plug-in.



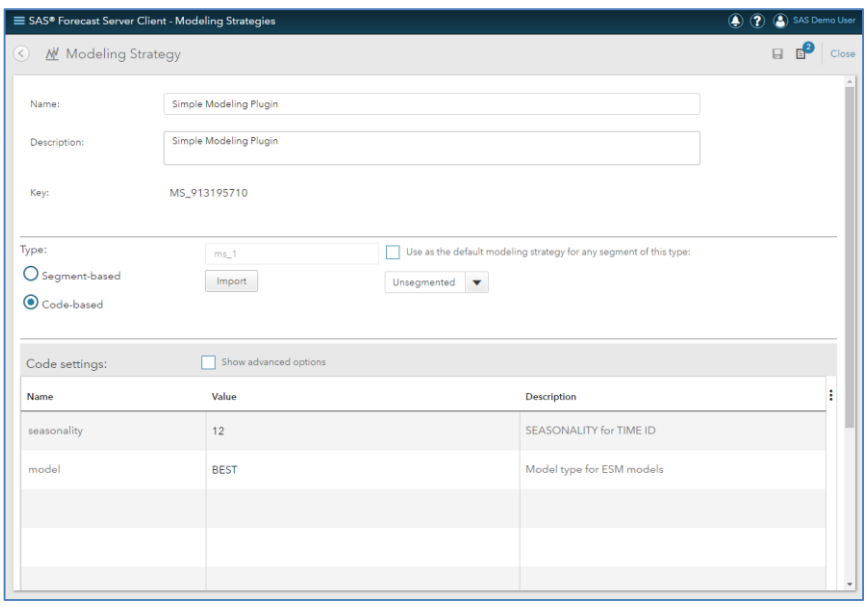
After the plug-in is registered, it is shown in the list of available segmentation strategies.



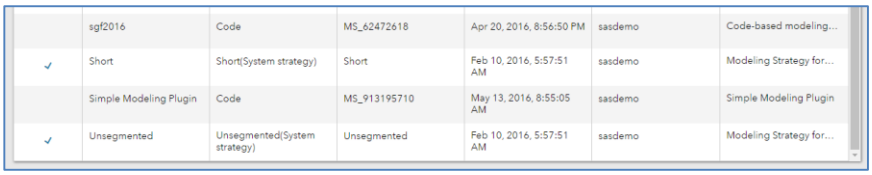
Similar to registering a segmentation plug-in, you can create a new modeling strategy based on the example plug-in in this document.



You can preview the settings provided by the plug-in.

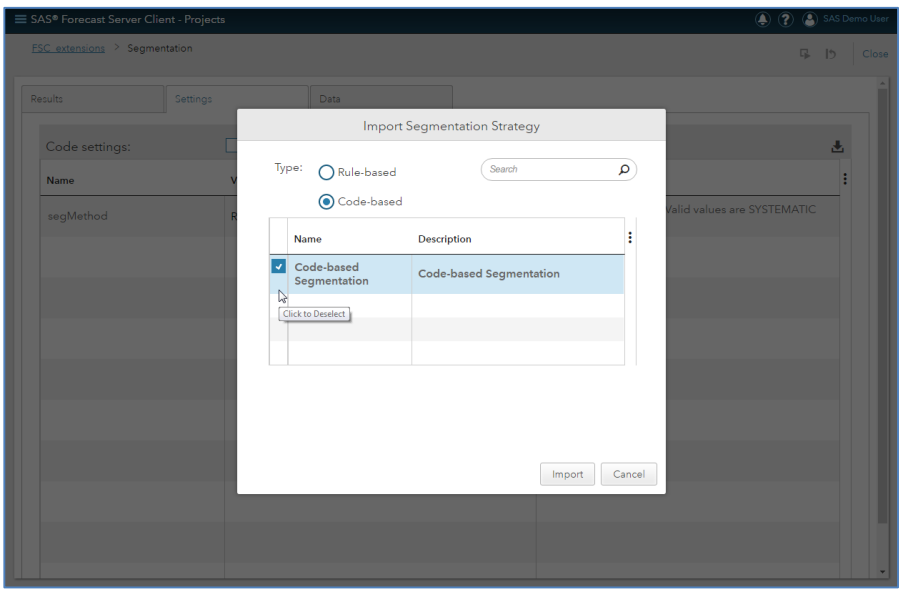


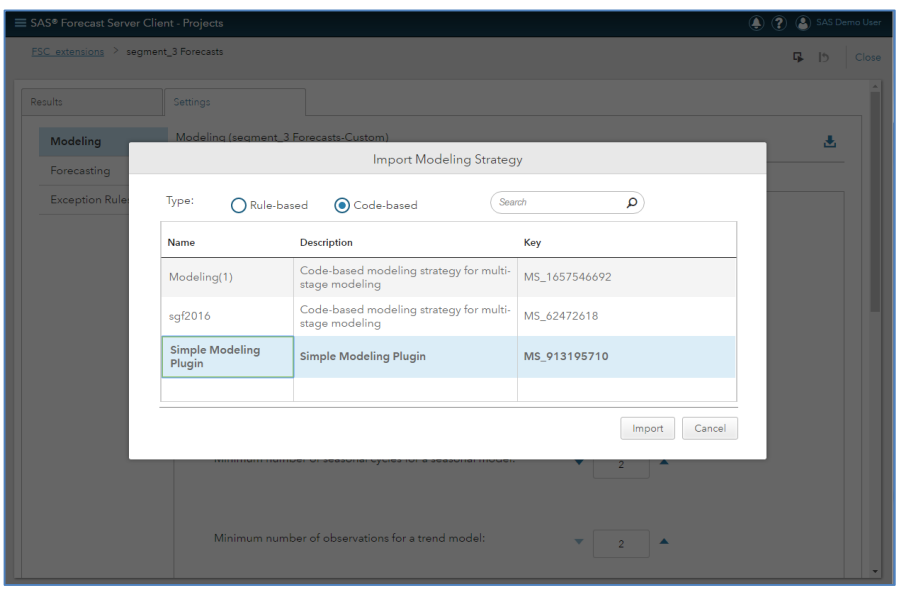
After the plug-in is registered, it is shown in the list of available modeling strategies, and a key is assigned to the newly created modeling strategy.



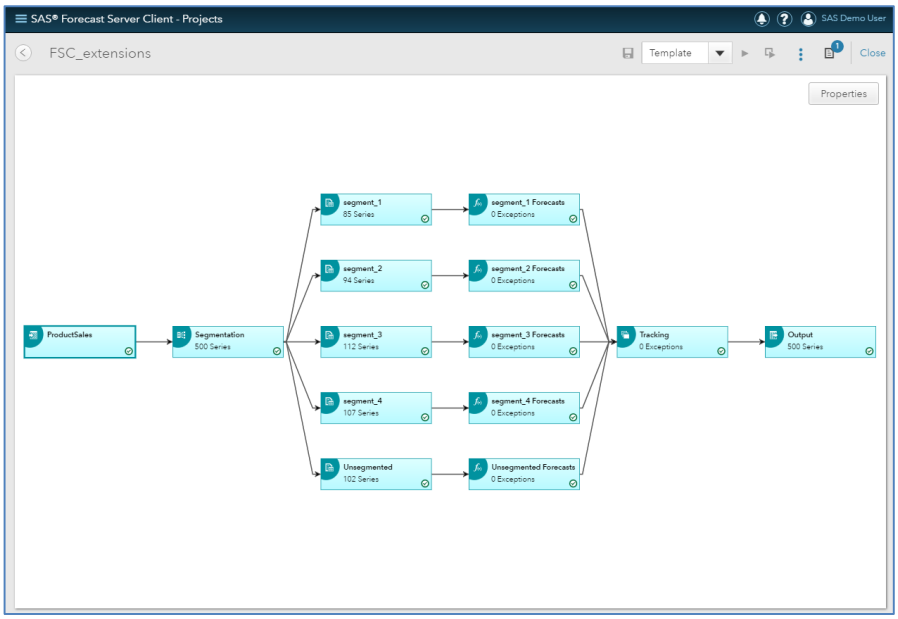
	sgf2016	Code	MS_62472618	Apr 20, 2016, 8:56:50 PM	sasdemo	Code-based modeling...
✓	Short	Short(System strategy)	Short	Feb 10, 2016, 5:57:51 AM	sasdemo	Modeling Strategy for...
	Simple Modeling Plugin	Code	MS_913195710	May 13, 2016, 8:55:05 AM	sasdemo	Simple Modeling Plugin
✓	Unsegmented	Unsegmented(System strategy)	Unsegmented	Feb 10, 2016, 5:57:51 AM	sasdemo	Modeling Strategy for...

You can now open a project and set the segmentation and modeling settings to use the registered plug-ins instead of the built-in strategies.

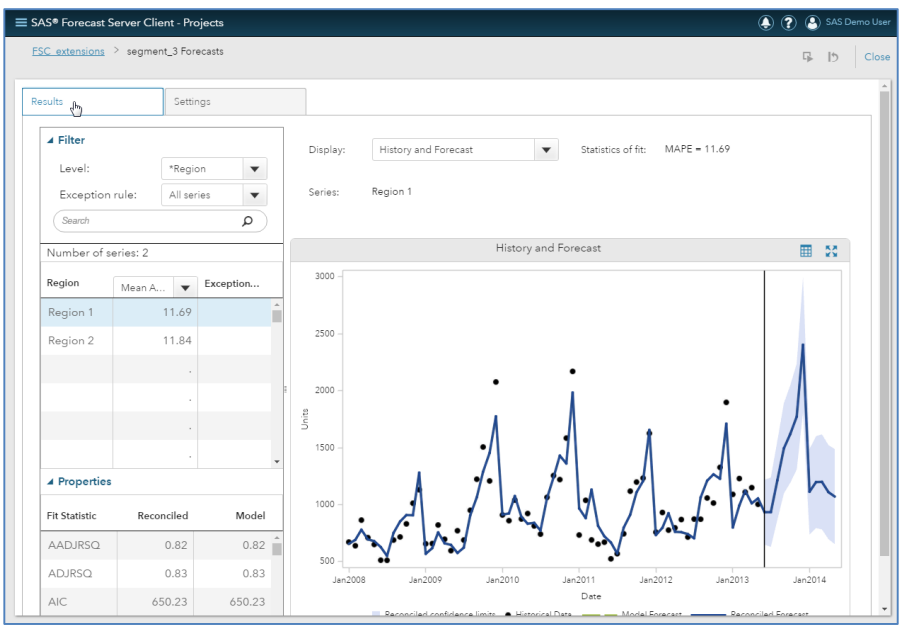




You can see in the project process flow diagram that the input data are segmented using the systematic segmentation method and that the simple forecast model is applied to each of the segments.



You can examine the forecast results from the modeling plug-ins.



Conclusion

SAS Forecast Server Client is a web-based interface included with SAS Forecast Server. It allows anywhere / anytime access, and integrates the capabilities from SAS Forecast Studio and SAS Time Series Studio. This paper has shown how to create “plug-ins” to further extend capabilities for such things as customized time-series segmentation and forecasting modeling strategies.

Resources

SAS® Forecast Server Client (demonstration video)
https://www.youtube.com/watch?v=LqpTCGeymos&feature=youtu.be

A Multistage Modeling Strategy for Hierarchical Demand Forecasting (SAS Global Forum Paper)
https://sasglobalforum2016.lanyonevents.com/connect/sessionDetail.ww?SESSION_ID=5260

“Forecast tracking by iteration with SAS Forecast Server Client”
http://blogs.sas.com/content/forecasting/2015/08/14/forecast-tracking-by-iteration-with-sas-forecast-server-client/

“Multistage modeling with SAS Forecast Server Client (Part 1)”
http://blogs.sas.com/content/forecasting/2015/08/13/multistage-modeling-with-sas-forecast-server-client-part-1/

“Multistage modeling with SAS Forecast Server Client (Part 2)”
http://blogs.sas.com/content/forecasting/2015/08/13/multistage-modeling-with-sas-forecast-server-client-part-2/

“Time series segmentation with SAS Forecast Server Client (Part 1)”
http://blogs.sas.com/content/forecasting/2015/08/11/guest-blogger-jessica-curtis-time-series-segmentation/

“Time series segmentation with SAS Forecast Server Client (Part 2)”
http://blogs.sas.com/content/forecasting/2015/08/12/time-series-segmentation-with-sas-forecast-server-client-part-2/

To contact your local SAS office, please visit: sas.com/offices