Model Developer's Guide: R Models

Introduction

The following will run through the process of developing an R-based model for the Analysis Services environment.

As an example, this tutorial will run through the process of developing an R-based model that computes the mean of a number of data streams.

Prior Reading

Although not essential, there are some other documents that are beneficial to read before embarking on this tutorial:

- The Analysis Services API Tutorial gives an overview of the application programming interface (API) for the Analysis Services environment.
- The Model Developer's Guide introduces the general concepts that are useful for all developers of models for the Analysis Services environment (i.e. not just those developing models in Python).

The R Code

The following R code defines a "multivariate mean" model and a number of helper functions:

```
library(httr)
library(rjson)

formatDateTime <- function(dt) {
    format(dt, format="%Y-%m-%dT%H:%M:%OS3%z")
}

doRequest <- function(method, config, path, query, ...) {
    url <- modify_url(config$url, path=gsub("/+", "/", paste(parse_url(config$url)$path, path, sep="/")))

    if ("apiKey" %in% names(config)) {
        query$apikey <- config$apiKey
        response <- method(url, query=query, ...)
    } else if ("username" %in% names(config) && "password" %in% names(config)) {
        response <- method(url, authenticate(config$username, config$password), query=query, ...)
    }

    stop_for_status(response)
```

```
}

storeObservations <- function(config, dataFrame, update, streamIds=NULL) {
    if (is.null(streamIds)) {
        streamIds <- colnames(dataFrame)[-1]
    }

    for (i in 2:ncol(dataFrame)) {
        results <- mapply(function(t, v) list(t=formatDateTime(t), v=list(v=v)), dataFrame[[1]], dataFrame[[i]], SIMPLIFY=FALSE)
        payload <- list(results=results)
        streamId <- streamIds[[i-1]]
        query <- list(streamid=streamId)

        r <- doRequest(POST, config, "observations", query, body=payload, encode="json", content_type_json(), accept_json())
        update(modified_streams=c(streamId))
    }
}

getObservations <- function(config, streamIds, start=NULL, end=NULL, si=NULL, ei=NULL) {
    query <- list(streamid=paste(streamIds, collapse=","), media="csv", csvheader=FALSE, limit=10000)

    if (!is.null(start)) {
        query$start <- start
    }
    if (!is.null(end)) {
        query$end <- end
    }
    if (!is.null(si)) {
        query$si <- si
    }
    if (!is.null(ei)) {
        query$ei <- ei
    }

    result <- NULL
    repeat {
        response <- doRequest(GET, config, "observations", query)

        data = content(response)
        result <- rbind(result, data)

        if (nrow(data) < query$limit) {
            break
        } else {
            query$start <- URLencode(formatDateTime(tail(data[[1]], n=1)), reserved=TRUE)
            query$si <- FALSE
        }
    }

    result
}

MultivariateMean <- function(context) {
    # Download observation data.
    context$update(message="Downloading observation data")
    data <- getObservations(context$sensor_config, context$ports$inputs$streamIds)

    # Compute mean.
    context$update(message="Computing mean")
    means <- apply(data[-1], 1, mean)
    result <- data.frame(timestamp=data['timestamp'])
    result[context$ports$output$streamId] <- means

    # Store results.
    context$update(message="Storing results")
    storeObservations(context$sensorConfig, result, context$update)
}
```

Besides importing the `httr` and `rjson` libraries, the core of the model is implemented in four functions. These are discussed in each of the following subsections.

The `MultivariateMean` Function

All R-based models must declare an "entry-point" function. These functions form the interface between the model and the rest of the Analysis Services architecture - the system executes the model by calling the function, passing in a context parameter representing the model's inputs, outputs and configuration data.

A simple convention is used to distinguish the entry-point function from any other functions present in the R code: the entry-point function must have the same name as the ID of the model it implements. In this example, the model's manifest (see "The Manifest" below) declares the model's ID to be `MultivariateMean`, hence the `MultivariateMean` function is this model's entry-point. Note that you can implement multiple models in a single R file by implementing multiple entry-point functions, each named according to the ID of a model declared in the manifest.

The entry-point function receives a single "context" parameter, which is a list with the following named members:

- `ports`, a list containing details of the model's inputs and outputs. See "The `ports` List" below.
- `sensor_config`, `analysis_config` and `thredds_config`, lists containing configuration details (URLs and credentials) for accessing the SensorCloud, Analysis Services and Thredds Data Server APIs. See "The `sensor_config`, `analysis_config` and `thredds_config` Lists" below.
- `update`, a callback function that can be called by the model code to provide realtime updates on the model's progress. See "The `update` Callback" below.
- `log`, a callback function that can be used to add entries to the model's log. See "The `log` Callback" below.

The `MultivariateMean` function uses these parameters to compute the mean of the input streams in three stages.

First, all the observation data for the input streams is downloaded. The list of input streams is provided as a character vector in the `streamIds` property of the input port. The `getObservations` function (see the corresponding subsection below) is used to do a `GET` request on the SensorCloud API's `observations` endpoint to download the data.

Then, the mean value across all the input streams at each timestep is computed by applying R's `mean` function across all columns of the obtained data frame but the first (which contains the observation timestamps). This returns a numeric vector containing the means, which is joined with the timestamps from the original data to create a new data frame representing the results.

Finally, the result is uploaded to the SensorCloud API using the `storeObservations` function (see below).

At each of these stages, a brief description of the model's progress is provided to the model execution framework by calling the `update()` callback with the message parameter provided. See "The `update` Callback" below for more detail.

The `ports` List

The `ports` list describes the model's inputs and outputs (known collectively as "ports"). These come in four kinds:

- "Stream" ports name single SensorCloud stream.
- "Multi-stream" ports name a number (zero or more) SensorCloud streams.
- "Document" ports contain a plain-text document.
- "Grid" ports describe a single Thredds dataset.

As well as having a type, a port has a name by which it is identified, and a direction (either "input" or "output"). Each of the ports declared in the model's manifest (see "The Manifest" below) is made available as a named property of the `ports` parameter, subject to one caveat: optional ports (those with `"required": false` in the manifest) are only made present in the `ports` parameter if actually supplied by the caller of the model. Each port in the `ports` parameter is a list with the following named properties:

- `name`: a string, the name of the port, as declared in the manifest.
- `direction`: a string, the direction of the port, either "input" or "output".
- `type`: a string, the port type, one of "stream", "multistream" or "document".
- For stream ports, `streamId`: a string, the ID of the stream to use as input/output.
- For multi-stream ports, `streamIds`: a character vector, the IDs of the stream(s) to use as input/output.
- For document ports, `document`: a string, the value of the document to use as input.
- For grid ports, `catalog`: the URL of the TDS catalog, and `dataset`: the relative path of the TDS dataset.

As a concrete example, the example model has two ports named "inputs" and "output" which could be obtained from the `ports` parameter as `ports$inputs` and `ports$output` respectively. The direction of the "inputs" port (i.e. `"input"`) can be obtained as `ports$inputs$direction`, its type (i.e. `"multistream"`) as `ports$inputs$type`, and the list of stream IDs associated with the port can be obtained as a character vector as `ports$inputs$streamIds`.

The `sensor_config`, `analysis_config` and `thredds_config` Lists

The `sensor_config`, `analysis_config` and `thredds_config` parameters each contain details (URLs and credentials) for accessing the SensorCloud, Analysis Services and Thredds APIs respectively. The configuration described by these parameters is adjusted dynamically to reflect the permissions of the user calling the model, as such it is best (but not absolutely required) to use these credentials rather than using credentials hard-coded into your model.

Each configuration consists of a number of named parameters:

- `url`: the "base" URL of the API (e.g. `https://senaps.io/api/sensor/v2/` for the SensorCloud API).
- `api_key`: an "API key" that can be added as a query parameter to requests in order to authenticate.
- `username` and `password`: a username and password that can be added to requests as a HTTP "basic authentication" header in order to authenticate.

Generally, only either the `api_key` will be supplied, or the `username` and `password` will be supplied, as these represent distinct authentication mechanisms.

Further detail of making requests to the APIs using these configuration details is beyond the scope of this document. Please refer to the Data Platform API Tutorial for details on how to access the SensorCloud API, or to the Analysis Services API Tutorial for details on how to access the Analysis Services API.

The `update` Callback

The purpose of the `update` callback is to provide a means for a running model to provide ongoing feedback to the model's caller as it is executing. It may be called at any time during execution of the model in order to update the model's status information. It accepts the following parameters, all optional:

- `message`: a string containing a brief textual description of the model's current status (e.g. "Downloading data"). If omitted, the model's status does not change - for example if one call to `update` specified `message="Downloading data"` and a subsequent call to `update` omitted the `message` parameter, then the model would still have a status of "Downloading data...". Pass `NULL` to indicate indeterminate status.
- `progress`: an indication of the model's progress towards completion, as a floating point number between `0.0` (no progress to completion) and `1.0` (model has completed). Intermediate values may be used to denote partial completion (i.e. `0.5` to indicate model is 50% complete). As with the `message` parameter, omitting the `progress` parameter will cause any previously provided value to be retained, and passing `NULL` may be used to indicate an indeterminate degree of progress.
- `modified_streams`: a character vector containing the ID of each stream which has had observations added or removed. This marks the listed streams as "modified", so that other models that depend on data in those streams may be re-run with the changed data. Once a stream has been marked modified using this parameter, it cannot be unmarked. Marking a stream modified more than once has no additional effect.
- `modified_documents`: a list with named parameters, each parameter name corresponding to the name of a "document" port, and each parameter value being a string containing the new value for that port's document. This is the means by which output document ports may have their values set. Documents may be marked modified multiple times, in which case only the latest document value is retained. There is no means to mark a document unmodified once it has been marked modified.

While the `message` and `progress` parameters are useful for providing helpful feedback to the caller of the model, the `modified_streams` and `modified_documents` parameters are crucial to the operation of each model, as it is by these parameters that the model's outputs are exposed.

The `log` Callback

The `log` callback provides a rudimentary logging facility for models running in R. It accepts the following parameters (all optional except message):

- `message`: a string, the log message.
- `level`: a string, the "log level" (i.e. severity). One of `"DEBUG"`, `"INFO"`, `"WARNING"`, `"ERROR"`, `"CRITICAL"`. If omitted, the message is logged with no particular level of significance implied.
- `file`: a string, the name of file generating the message (e.g. "my_model.r").
- `line`: an integer, the line number the message pertains to (if any).
- `timestamp`: a string, containing the message's timestamp in ISO-8601 date/time format. If omitted, the current time is used.

The `doRequest` Function

The purpose of the `doRequest` function is to make an HTTP request to an API, using a set of supplied configuration options (see "The `sensor_config`, `analysis_config` and `thredds_config` Lists" above). It's thin wrapper around `httr`'s various HTTP functions (`GET`, `POST`, etc). It takes four main parameters:

- `method`: the `httr` request function to wrap (e.g. `GET`).
- `config`: the API configuration, as described in "The `sensor_config`, `analysis_config` and `thredds_config` Lists".
- `path`: the resource path in the API (e.g. `"observations"` to access the SensorCloud `/observations` endpoint).
- `query`: the query parameters for the request.

In addition to these, the function accepts any number of additional parameters that are passed through unchanged to the `httr` request function specified in the method parameter.

This function uses the `url` property of the supplied configuration and the `path` parameter to compute the complete URL of the endpoint of interest. Then, if an API key is present in the configuration, it adds the appropriate query parameter. Otherwise, if a username and password is specified in the configuration, it adds an HTTP basic authentication header to the request. It then passes responsibility to the requested `httr` request function.

The `getObservations` Function

The `getObservations` function is used to retrieve sensor observation data from the SensorCloud API. The function takes the following parameters:

- `config`: the configuration details of the SensorCloud API to retrieve the observations from (see "The `sensor_config`, `analysis_config` and `thredds_config` Lists" above).
- `streamIds`: a character vector containing the IDs of the streams to retrieve observations for.
- `start`: an optional ISO-8601 formatted date/time indicating the start of the temporal window to retrieve. If not specified, the temporal window begins at the date/time of the earliest observation.
- `end`: an optional ISO-8601 formatted date/time indicating the end of the temporal window to retrieve. If not specified, the temporal window ends at the date/time of the latest observation.
- `si`: an optional Boolean indicating if the `start` parameter is an inclusive (if `TRUE`) or exclusive (if `FALSE`) boundary. Defaults to `TRUE` if not specified.
- `ei`: an optional Boolean indicating if the end parameter is an inclusive (if `TRUE`) or exclusive (if `FALSE`) boundary. Defaults to `TRUE` if not specified.

Given these parameters, all the corresponding observations are retrieved from the SensorCloud API. A sliding window 10,000 observations wide is used to ensure that all the observations are retrieved. The results are returned as a data frame, with the leftmost column containing the timestamps as `POSIXct` objects (https://stat.ethz.ch/R-manual/R-devel/library/base/html/DateTimeClasses.html), and the remaining columns containing the observation data.

The `storeObservations` Function

As the name suggests, the `storeObservations` function can be used to store an R data frame as a (number of) time series in the SensorCloud API. In order to do so, the data frame must have the timestamps of the time series in the leftmost column, as `POSIXct` objects. There must also be at least one more column containing scalar values to use as the observation values at each time step.

By default, the function assumes that the column names for the data columns correspond to the stream IDs to store the observations into. Should that not be the case, the stream IDs can be overridden by supplying them as a character vector to the `streamIds` parameter.

The full set of parameters to the function are as follows:

- `config`: the configuration details of the SensorCloud API to store the observations into (see "The `sensor_config`, `analysis_config` and `thredds_config` Lists" above).
- `dataFrame`: the data frame to store.
- `update`: the `update` callback, passed through from the model's entry-point function. Used to mark the relevant streams as modified.
- `streamIds`: an optional character vector containing the IDs of the streams to use (in preference to the data frame's column names).

The function iterates over each of the data columns, converts the data to the format expected by the SensorCloud API, and then executes a `POST` request to the `observations` endpoint. Finally, it marks the corresponding stream as modified, so that other models relying on the data may be re-run as necessary.

NOTE: Installing Libraries

In the above code, it's worth noting that (unlike typical R code) the `httr` and `rjson` libraries were unconditionally loaded without any additional logic to install the libraries if absent. That is, where the above code has just `library(httr)`, a common R idiom would look more like the following:

```
if(!require(httr)){
    install.packages("httr")
    library(httr)
}
```

The reason this approach is not used in the above model code is because running models are isolated from the internet (for security reasons), and therefore cannot install libraries at runtime. Any libraries that the code requires must either be declared in the model's manifest (see "The Manifest" below), or must be included in the model's base image. In this particular case, the httr and rjson libraries are pre-installed in the model's base image.

The Manifest

The following JSON document is the model's manifest:

```
{
    "baseImage": "da9b54b0-5467-4a9f-b59a-58525885ddc5",
    "organisationId": "csiro",
    "groupIds": [],
    "entrypoint": "model.r",
    "dependencies": [],
    "models": [
        {
            "id": "MultivariateMean",
            "name": "Multivariate Mean",
            "version": "0.0.1",
            "description": "Computes mean of multiple aligned data streams.",
            "method": "",
            "ports": [
                {
                    "portName": "inputs",
                    "required": true,
                    "type": "multistream",
                    "description": "The streams to be averaged",
                    "direction": "input"
                },
                {
                    "portName": "output",
                    "required": true,
                    "type": "stream",
                    "description": "The stream to place the averaged data into.",
                    "direction": "output"
                }
            ]
        }
    ]
}
```

For a detailed discussion of the format of a manifest, please refer to "The Manifest File" in the main Model Developer's Guide.

In this specific case, the manifest declares the following:

- The model image will be based on the "R-Base" image (with base image ID da9b54b0-5467-4a9f-b59a-58525885ddc5).
- The model will be "owned" by the CSIRO organisation.
- The model is not "owned" by any groups within the CSIRO organisation.
- The main model code is in the file "model.r"
- The model has no additional third-party dependencies (all required dependencies in this case come pre-installed on the selected base image).
- There is a single model implemented:
  - The model has the ID MultivariateMean, corresponding to the name of the entry-point function implemented in the R code.
  - The model's human-friendly name is "Multivariate Mean".
  - The model's version is 0.0.1.
  - A description of the model, and a summary of the approach it uses (i.e. the "method") are also provided.
  - The model has two ports:
    - The "inputs" port is a multi-stream input port that takes a list of the streams to be averaged.
    - The "output" port is a stream output port that takes the ID of the stream to place the computed average into.

Installing the Model

The generic process for installing a new model is discussed in detail in the "Installing Models" section of the Model Developer's Guide.

In summary, installing the example model requires placing the R code file and the manifest file into a ZIP or tar/gzip archive file, and uploading it to the Analysis Services' `models` endpoint with a `POST` request.