Paper SAS1882-2018

restAF – A JavaScript Library for Rapidly Developing SAS® Viya® Applications Based on SAS® REST APIs

Deva Kumar, SAS Institute Inc.

ABSTRACT

SAS® Viya® is like a Swiss Army knife when it comes to allowing all types of clients to talk to it—Java, Python, Lua, R, and, of course, SAS®. In addition to using these language-based APIs, SAS also publishes APIs that conform to REST best practices. The well-designed APIs in SAS Viya make it relatively easy to build complex web and non-web applications with SAS Viya as the engine of your application. The applications can be solely based on SAS Viya capabilities or users can do a mashup of the advanced SAS Viya analytics with their other products and capabilities. restAF is a light-weight, UI-framework agnostic, JavaScript library designed to be easy-to-use for app developers. restAF takes advantage of the consistency of the REST APIs to reduce the responses from the server into a standard form, and it also manages the data on the client side. This reduction enables developers to interact with the server in a repeating pattern regardless of which service the application is using. In this presentation, we explain the concepts behind restAF and demonstrate building applications that exploit SAS® Cloud Analytic Services (CAS), compute server, SAS® Visual Analytics, and other key capabilities of SAS Viya in a very consistent manner with minimal coding. restAF (along with all the examples and detailed documentation) are available as an open-source project. You can download it from GitHub with the usual open-source licensing policies.

INTRODUCTION

The goal of restAF is to provide a simple programming model to access SAS Viya using REST APIs.

1. restAF provides a small set of promise-based methods to make the API calls.
2. restAF automatically “reduces” the returned information to a form that simplifies using it in the application.
3. restAF manages the application data that can be accessed anywhere in your application as the single version of the truth.

With restAF, you focus on your application and not on the nitty-gritty details of setting up HTTP calls, processing the returned results and HTTP codes, parsing the returned payload, managing the data, and so on.

> **Note**: A far more detailed document is available in the https://github.com/sassoftware/restaf repository.

OVERVIEW

A typical response from a REST API is a hypermedia that allows applications to decide what can be done next – either with user input or programmatically. In practical terms, a response from the server will be a combination of items (data) and transitions (links to navigate) to another resource.

LINKS

A link has two key parts:

1. A URI for the next step (edit, delete, and so on).
2. An identification of the link to indicate what this next step is. This is referred to as “rel” – short for “link relationship.” You can discern the purpose of that link based on the rel. If the rel is “content,” you will guess that it returns the content. If the rel is “execute,” you will guess that this link is associated with some type of execution that is appropriate for that context. If the rel is “create,” you will rightly assume that the call will result in something being created.

ITEMS

Items are the data. Items can be a collection of items, a text string, an array, SVG, and so on.

Some examples are as follows:

1. A collection of items: Each item has some data and links to indicate what you can do with this item (delete for example).
2. A string (examples: SVG, status of a job, and so on).
3. An array (example: SAS logs).
4. Some object.
5. And, in some cases, no data.

MEDIA TYPE

Each of the responses comes with a media type. An application can (should?) use this information to handle the returned data appropriately. An example would be choosing the appropriate viewer for the returned data.

SO WHY restAF?

restAF takes advantage of the consistency of the REST APIs to reduce the responses from the server into a standard form, and also manage the data on the client side. This reduction enables developers to interact with the server in a repeating pattern regardless of which service the application is using. This, in turn, will reduce the development and testing times.

BASIC FLOW OF AN APPLICATION USING restAF

Below is a short description of four basic steps you will use to write your application:

1. **start** - Initialize restAF using the initStore method of restAF and obtain the “store” object. This object is used to access the server via REST calls and to track all the returned data.
2. **connect store to server** – Use the logon method of the store to connect the store to the SAS Viya server. The “payload” argument has information about the server and authentication. restAF supports the Oauth2 authentication supported by SAS Viya.
3. **Add services to store** – Tell the store which services you want to access via the addServices method of the store. You can add multiple services with one call or add them as your application needs them. This lets the store set up internal folders to manage the data for these services.
4. **Main loop of your application** - In the main loop, your application will ask the store to make REST calls for your application. When the server returns a response, the store “reduces” it to an internal format and returns an object called the rafObject. This rafObject has methods through which you can access the links and items from anywhere in your application. The data in the store is Read-Only. This ensures that in any part of your application you are getting the “truth” at all times.

Note: While the main purpose of restAF is to act as an intermediary between your application and the SAS Viya Server, it can also store your application-specific data.

PROMISES, PROMISES, PROMISES

restAF returns promises from methods that make asynchronous calls to the server. So become familiar with promises. While you can stay with the standard handling of promises, I strongly recommend that you use the new async functions in ES2017.

The document will use both pre-ES7 and ES7 syntax.

INTRODUCTORY EXAMPLE – NODEJS APPLICATION

The example code is all the code you will need to run a DATA step with the compute service. The next few sections will explain how this code works and, in the process, introduce you to the various key aspects of restAF:

```
   // See the examples repository for the code
let restaf   = require('restaf');
let config = require('./config');
let logonPayload = config('raf.env');


// step 1
let store        = restaf.initStore();

// steps 2 and 3
async function setup (store, payload, service){
    let msg  = await store.logon(payload);
    let root = await store.addServices(service);
    return root;
}

// Step 4
async function mainLoop (store, compute, code) {

    // get the list of contexts for compute server
    let contexts = await store.apiCall(compute.links('contexts'));

    // lookup the name of the first context,
    // use it to get the associated createSession rafLink
    // create a compute session
    let context       = contexts.itemsList(0);
    let createSession = contexts.itemsCmd(context, 'createSession');
    let session       = await store.apiCall(createSession);

    // Define the payload
    let payload = {
            data: {code: code}
    };

    // Execute the data step and wait for completion
    let job    = await store.apiCall(session.links('execute'), payload);
    let status = await store.jobState(job, null, 5, 2);
```

```
        // Check the final status of job
        if (status.data === 'running') {
            throw `ERROR: Job did not complete in allotted time`;
        } else {
            let logLines = await store.apiCall(status.job.links('log'));
            // print the log
            logViewer(logLines);
        }
        return 'restAF is cool or what';
    }

    // Your main app

    let code =  [`data _null_; do i = 1 to 100; x=1; end; run; `];

    setup (store, logonPayload, 'compute')
        .then (compute => mainLoop(store, compute, code))
        .catch(err =>console.log(err));
```

INTRODUCTORY EXAMPLE – WEB APPLICATION

You can reuse the code above to write a simple web application. Below is the code for a simple web application where you enter the SAS code and view the logs. The simple app is shown in Figure 2.

As you look at the code notice that the setup function and the mainLoop functions are exactly the same as in the nodejs application. An immediate benefit of reusing the code is that you can debug key aspects of your code in a nodejs application first and then incorporate it in your web application.

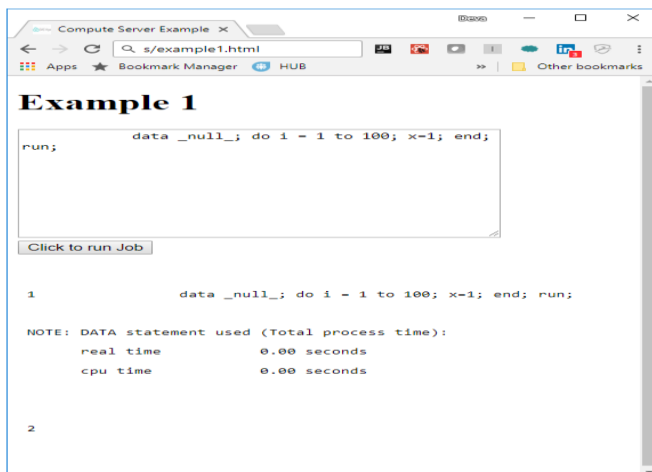


Figure 1. Web Application for Introductory Example

The code for the web application is as follows:

```
<script type="text/javascript">
// Setup
let store    = restaf.initStore();
let compute  = null;
let onClickb = onClick.bind(this);
```

```
async function setup (store, payload, service){
    let msg  = await store.logon(payload);
    let root = await store.addServices(service);
    return root;
}

// Main Loop
async function mainLoop (store, compute, code) {


// get the list of contexts for compute server
let contexts = await store.apiCall(compute.links('contexts'));

// lookup the name of the first context,
// use it to get the associated createSession rafLink
// create a compute session

    // Define the payload
    let payload = {
        data: {code: code}
    };

    // Execute the data step and wait for completion
    let job    = await store.apiCall(session.links('execute'),payload);
    let status = await store.jobState(job, null, 5, 2);

    // Check the final status of job
    if (status.data === 'running') {
        throw `ERROR: Job did not complete in allotted time`;
    } else {
        let logLines = await store.apiCall(status.job.links('log'));
        // print the log
        logViewer (store, logLines);
    }
    return 'restAF is cool or what';
}

// Execute
function initialize() {
    let logonPayload = config();
    debugger;
    setup(store, logonPayload, 'compute')
        .then(root => {
            compute = root;
        })
        .catch(err => showAlert(err));
}

// visuals
function onClick() {
    let codeText = document.getElementById('pgm').value;
    let code     = codeText.split(/\r?\n/);
    mainLoop (store, compute, code)
        .then (msg => msg)
        .catch(err => showAlert(err));
}
```

```
</script>

// Your HTML code
<body onload="initialize()">
<h1> Log from Compute Server Job using restAF</h1>

<div>
    <form>
        <textarea id="pgm" rows="10" cols="50">
            data _null_; do i = 1 to 100; x=1; end; run;
        </textarea>
        <br>
        <input type="button" value="Click to run Job"
onclick="onClickb()">
    </form>
    <br>
    <br>

    <form class="info">
        <pre id="log"></pre>
    </form>
    <br>
</div>

</body>
```

TERMINOLOGY

store – object to manage access to the SAS Viya server and the resulting application data. Store can also be used to manage your applications data. The data in the store is immutable. The data is updated only through methods of the store.

rafObject – restAF reduces the information returned by an API call to a form that makes it convenient for the application to consume. The returned object is called a rafObject. You will use this to retrieve information from the store.

rafLinks – restAF transforms all returned links into an object called rafLinks. restAF uses rafLinks to navigate its own structures and make API calls on behalf of the application. rafLink is an opaque object that you will get through rafObject methods and then use it in the restAF methods for api calls.

rel – The link relationships that are the key to using restAF.

ACCESSING THE SERVER

This section will use the example shown earlier to explain how restAF works.

Initializing restAF

To initialize restAF, you will make the following call:

```
let store = raf.initStore();
```

This initializes the store object. Only one store per application is allowed. At this point, the store is empty and has no connection to a Viya Server.

So the next step is to connect restAF to a Viya Server using the logon method.

Logon

The logon method takes a logon payload:

```
store.logon( logonPayload )
  .then ( status => { … run your app …}
  .catch( err => {… handle logon failure…}
```

The logonPayload has information about how restAF should connect to the server. restAF supports OAuth2 password and implicit flows. If logonPayload is null, restAF assumes that your browser session has been authenticated by other means. Otherwise, the logonPayload has information to inform restAF on how to handle authentication. Please refer to the document in the repository for the details about logonPayload.

The next step is to let restAF know what services you plan to access.

Adding Services

addServices method adds a folder for each service. It also calls the root end point for that service and populates the folder with the links returned from that call. The addServices method returns a rafObject. You are now ready to use this service in your application.

```
let rafObj = await store.addServices (serviceName);
… your app does what it needs to do.

Alternatively:

store.addServices (serviceName)
    .then (rafObj => {
          //your app does what it needs to do
    } )
    .catch ( err => {
      //handle error condition
    } )
```

Adding Multiple Services

If your application uses more than one of the services, you can add all of them by passing a list of services to the addServices method. If you pass more than one service, the returned value is an object with the rafObjects representing the services that were started. For example, if you wish to add casManagement and compute service, then the code will look as follows:

```
let rafObjects = await store.addServices( 'casManagement', 'compute' )
let casManagement = rafObjects.casManagement;
let compute       = rafObjects.compute;
      …do something…
```

Making API Calls

The apiCall method on the store object is the key method for making REST calls. The syntax is as follows:

```
    store.apiCall( rafLink <,payload> )
    .then (rafObject => {…})
 .catch (err => {…})
```

Other methods you will use to access the server are apiCallAll, submit, jobState, and jobStateAll. Please see the full documentation for details about these.

Payload

The payload (optional parameter) is an object with the following optional keys:

data - If the REST call is a PUT or POST, then specify the data payload.

qs – Use this to specify the query parameters.

headers - Use this to specify any special headers for this call. For example, the upload action in cas requires the JSON-Parameters header. Another example is file service's 'create' operation, which requires content-type to be specified.

action - If you are running a cas action, you **must** specify the action name. We recommend that you use a fully qualified action name (actionset.actionname).

All other keys will be ignored silently.

Examples of Payloads

```
To run a datastep in compute service:
   {data: { code: [ 'data a; x=1;run;' , 'proc print;run' ] }}

To run datastep action in CAS:
   {action: 'datastep.runCode', data: { code: 'data a; x=1;run;' } }

To run upload action for CAS:
   let JSON_Parameters = {
                 casout: {
                     caslib: 'casuser', /* a valid caslib */
                     name  : 'iris' /* name of output file on cas server */
                 },

                 importOptions: {
                     fileType: 'csv' /* type of the file being uploaded */
                 }
         };
    let payload = {
         action : 'table.upload',
         headers: { 'JSON-Parameters': JSON_Parameters },
         data   : readFile( 'iris', 'csv' )
     }


To create a file with plain text:
   let payload = {
       data   : 'my data',
       headers: { 'content-type': 'text/plain' }
```

```
}
```

DETAILS ON RAFOBJECT

This section explains the rafObject and how to use it.

PROPERTIES

All property names are case sensitive. The key ones are documented here. Please see the repoDoc for more.

type: This indicates which of the results patterns this object represents.

1. **links:** Use the links method below to view and obtain rafLinks for further calls. If type is "links," then this object only has links for other end points.
2. **itemsList:** The rafObj contains an array of items with IDs.
    a. Use the itemsList method to retrieve the IDs of the items returned.
    b. Use the scrollCmds method to determine whether there is more data available and for making the pagination calls.
    c. Use the items function to retrieve specific items (or all items) from the collection.
    d. Use itemsCmd to retrieve commands (links) associated with specific item (examples: delete, copy, content, and so on).
3. **itemArray:** The returned value is an array of text. Use the items methods to retrieve this data.
4. **Items:** The exact object type is not known. Use the items method to get the information and also use the resultType property to decide how to handle that data.

resultType: The media type of the result.

status: This is the HTTP code returned from the API call.

statusInfo: An object with details of the status.

METHODS

Use these methods to interact with rafObject. These methods return either objects or string depending on the data. Remember you should not modify the returned data. In the document below, the common usage patterns are described.

You can use query parameters to drill into a specific value in that object. Examples are provided below.

links method

The links method will return an immutable object of rafLinks. You use this method when the type of the rafObject is "links." But sometimes there will be links even when the type is not "links." In those cases, the links are usually actions that operate on the whole collection. The syntax is as follows:

```
rafLink = rafObj.links( relName );
```

relName is the rel of the resource you are looking for. The call will return a rafLink that you can use in an apiCall as shown below:

```
contentLink  = fileObject.links('content');
```

```
contentObject = store.apiCall (contentLink);
```

Sometimes you need the entire list of links. An example is displaying buttons with the rel of the links as labels. Below is an example of printing the titles of all the links:

```
  allLinks = fileObject.links();
allLinks.forEach((l, rel) => console.log(`rel: ${rel} `);
```

itemsList method

If the rafObject type is 'itemsList', use this method to get the array of IDs in the current result object.

Below is an example listing all the IDs:

```
let idList = rafObj.itemsList();
idList.map( id => console.log(id) );
```

scrollCmds method

This method returns the rafLinks associated with pagination. This method can be used to get a rafLink associated with the next, prev, last, and first rels. An API call to the server might return some, all, or none of these rels. To ensure safe programming, always check if the returned rafLink is non-null. A sample code is as follows:

```
  let nextCmd = rafObj.scrollCmds( 'next' );
if ( nextCmd !== null ) {
   store.apiCall( nextCmd )
   .then ( rafobject => {...do something...} )
  } else {
      console.log( 'No more data' )
  }
```

items method

This method gives you access to the data that might have been returned. This could be a log for compute server, CAS results, tables, status, ODS output, and so on. The combination of resultType and type should give you sufficient information on how to process this data. It is possible to write an intelligent UI that determines the "viewer" to use given these pieces of information.

The items method takes arguments that determine what is returned:

```
let result = rafObj.items( what-you-want )
```

Let us examine a few typical cases.

To get all items:

```
let allItems = rafObj.items();
```

If you get all the items, then you need to write the code to manage the items based on the resultType.

Get a specific item using the ID that you got from itemsList method:

```
let item = rafObj.items(idOfItem);
```

itemsCmd

Use this method to obtain the commands associated with a particular item. Obtain the ID of the item by using the itemList method.

Get all commands associated with an item with a specific ID:

```
let item = rafObj.itemsCmd( idOfItem );
```

You can step through this object as follows:

```
rafObj.itemsCmd( idOfItem ).forEach( ( c, key) => {
 // c is the current cmd information
 // key is the name of the command( createSession delete etc... )
 }
```

Get a specific command associated with an item with a specific ID:

```
let item = rafObj.itemsCmd( idOfItem, 'name of command' );
```

Below is an example of obtaining the delete rel and then deleting the item:

```
let deleteCmd = rafObj.itemsCmd( '1234ID', 'delete' );
 store.apiCall( deleteCmd )
    .then ( f => { do whatever) }
    .catch( err => { do whatever) }
```

responseHeaders

On occasion, you might need to access the returned headers like etag. For those scenarios, use the responseHeaders method.

Get a specific header:

```
let etag = rafObj.responseHeaders( ‘etag’ );
```

Get all headers:

```
let headers = rafObj.responseHeaders();
```

status

To get the HTTP status returned from the call:

```
let status = rafObj.status();
```

statusInfo

Use this to get any additional information that might have been returned for the status. This is useful mostly if the job failed for some reason. An example is as follows:

```
let info = rafObj.statusInfo() ;
console.log (info);
```

A SIMPLE WALK-THROUGH WITH FILE SERVICE

The files service is a good use case for understanding how to use restAF. The discussion below assumes that you have already initialized the store and logged on to a Viya Server.

STEP 1: ADDING FILES SERVICE TO STORE

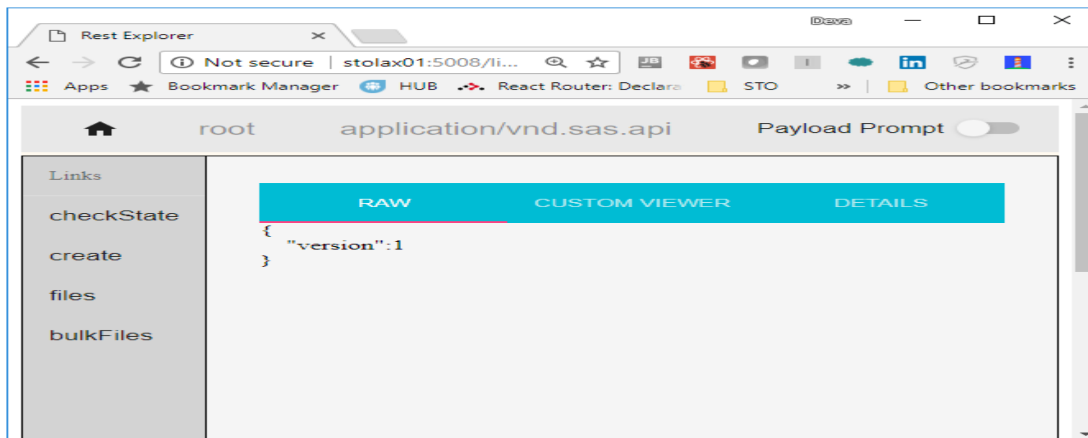
The code is as follows:

```
let filesObj  = await  store.addServices('files');
```

At this point, we can list the rels returned by the root. To do that, we use the links method to get the rels.

```
filesObj.links().forEach((l, k) => {
  console.log(k);
})
```

The figure below shows the root links returned by the addServices call.



STEP 2: GETTING THE LIST OF FILES

From documentation, we know that the "files" rel returns a list of files. Since this list can potentially be large, the service returns a small set and links for navigation. To make the store.apiCall, we need to get the rafLink for the "files" rel. For this, we use the links method and pass it the rel (=files) as a query parameter.

```
let rafLink = filesObj.links('files');
```

Then use the apiCall method to get the list of files

```
let filesList = await store.apiCall(rafLink);
```

At this point the filesList object has information on the first set of items returned from the call.

filesList.links() might return some links like 'self'.

filesList.scrollCmds() will return the pagination links (next, prev, first, and last) if they are appropriate.

filesList.itemsList() will return an array of the IDs of the returned item. fileList.itemsList().size will give you the count on the items.

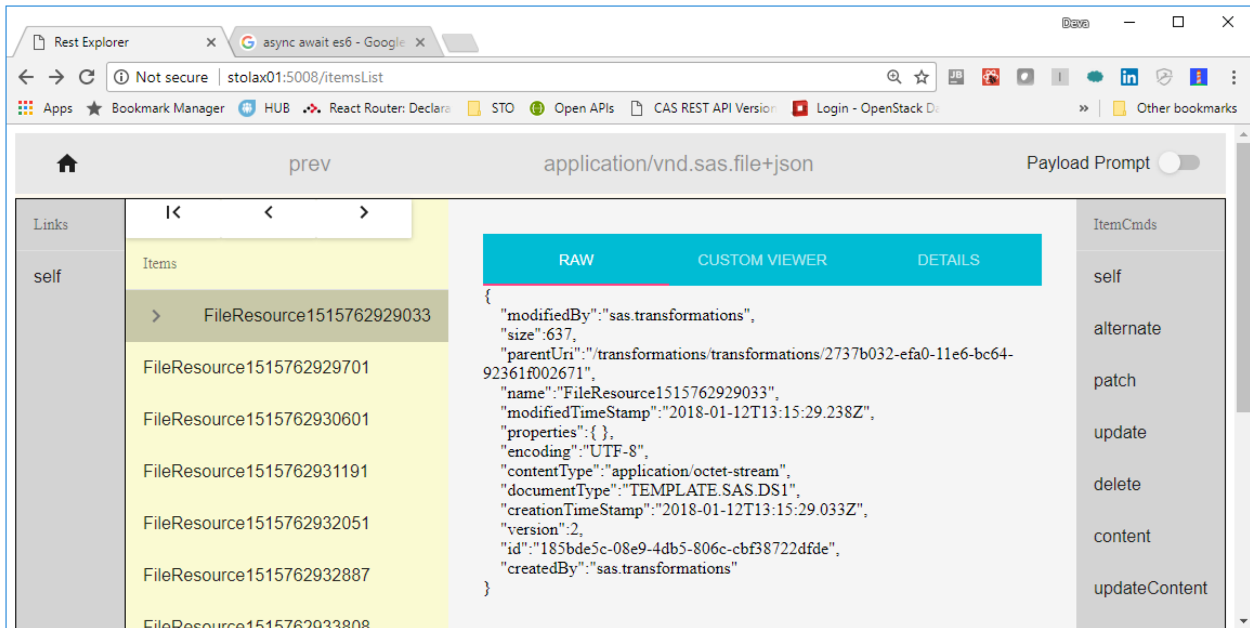
filestList.itemsList(n) will return the nth ID – let us call it fileId.

filesList.items(fileid) will return the summary information on the specified filedId.

Each of these items will have additional links that will enable you to operate on that particular item. The rels for each item can be obtained with the following code:

```
filesList.itemsCmd(filedId);
```

The figure below shows the details discussed above. The first panel shows the links returned by the call. The second panel shows a list of items. The pagination links (scrollCmds) are displayed as arrows in that panel. The next pane is the summary information on the selected item in the second panel. The last panel represents the links associated with the selected item. You can think of the ItemCmds as “context menus.”



STEP 2: OPERATING ON A SPECIFIC FILE

To do something with a specific item (example: the selected item in the figure above), you will do the following:

To print the contents of a file to the console:

```
    let rafLink = fileList.itemsCmd(fileId, ‘content’);
let fileContent = await store.apiCall(rafLink);
console.log( JSON.stringify(fileContent.items(), null, 4);
```

To delete the file:

```
let rafLink = fileList.itemsCmd(fileId, ‘delete’);
let fileContent = await store.apiCall(rafLink);
```

To create a new file:

```
let payload = {
    data    : 'my data',
    headers: { 'content-type': 'text/plain' }
 }
let rafLink   = fileList.itemsCmd(fileId, 'create');
let newFile = await store.apiCall(rafLink, payload);
```

At this point, you get the general idea. As you write more code, you will see that this pattern repeats. And over time, you will create some wrapper functions to further simplify your code. The setup and mainLoop functions in the first example are perfect examples of re-useable functions you might develop.

In the next few sections, key usage scenarios are discussed. Please see the repositories for complete examples.

USEFUL CODING SCENARIOS

EXECUTING CAS ACTIONS

restAF wrappers CASactions by adding an 'execute' rel to sessions created via casManagement.

The steps are as follows:

1. Create a session with casManagement..

   The example below shows how to execute a CAS action

```
   let root = await store.addServices('casManagement');

   //Get list of current servers
   let servers = await store.apiCall(root.links('servers'));

   //Create a session on the first server
//that is your only option in Viya 3.3 since there is only one cas
server
let casserver = servers.itemsList(0);
let session = await store.apiCall(servers.itemsCmd(casserver,
'createSession'), {data: {name: "somesessinmame}});
```

2. Get the execute rafLink for CAS actions.

```
let actionRafLink = session.links ('execute');
```

3. Create the payload for apiCall

   In the payload, specify the action name as shown in the example below. We recommend you use the fully qualified name of the action—it's just good programming practice. Below is an example:

```
let p = {
    action: 'datastep.runCode',
    data  : { code: 'data casuser.score; x1=10;x2=20;x3=30; score1 =
x1+x2+x3;run; '  }
```

```
};
```

4. Execute actions using apiCall.

   . Below is an example of running a CAS datastep and checking the results:

```
let actionResult = await store.apiCall(actionRafLink, p);

//Check the status of the result
let statusCode = actionResult.items('disposition', 'statusCode');
if (statusCode !== 0) {
    throw actionResult.items('disposition');
} else {
    prtUtil.view(actionResult, 'DataStep action');
}
```

HANDLING TABLES RETURNED BY CAS

restAF collects all the tables returned by CAS into an object in items with a key of “tables”.

To get to tables, use the snippet of code below:

```
let tablex = actionResult.items ('tables', 'name of table');
```

To get a list of the table names use the following snippet:

```
actionResult.items('tables').forEach( (t, name) => {
   …do whatever…
})
```

USING MEDIA TYPE

The responseType property of a rafObject is the media type of the response. Below is a sample react component code that uses the resultType to determine the viewer to use. There is probably a similar pattern in the UI framework that you use.

```
function getCustomViewer ( rafObject ) {

     let resultType = rafObject.resultType;
     let Viewer = null;
     switch (resultType) {
         case 'application/vnd.sas.compute.log.line':
             Viewer = <LogLines folder={rafObject}/>;
             break;
         case 'application/vnd.sas.compute.data.table.row.set':
             Viewer = <ComputeDataTableRowSet folder={rafObject} />;
             break;
         default:
             Viewer = <h2> No custom viewer at this time </h2>;
             break;
     }

     return Viewer;
}
```

LISTING ITEMS

If the rafObject type is “itemsList,” you can iterate over the list of items to create a display. Below is a function (using react) to display the items in a list. The map function is used to iterate over the returned list of items.

```
makeTable( folder ) {
    let itemsList = rafObject.itemsList();
    return <div className="idTable"><ul>
             { itemsList.map((m, i) => <li key={i}> {m} </li>) }
             </ul>
           </div>;
 }
```

CREATING PAGINATION MENUS

The function below creates a menus for scrolling using react. The key in the function is the forEach loop that iterates over the pagination links that might have been returned. Try this in your own UI framework. (Note: onClick is an array of event handler for the buttons defined elsewhere in your program and is not part of restAF.)

```
    let {onClick} = <array of handlers for the button onClick event>
    let cmds = rafObject.scrollCmds();
    let menus = [];
    cmds.forEach((c,rel) => {
        menus.push(<button key={rel} onClick={onClick[rel]}
                  className="button"> {rel} </button>);
    });
}
```

LISTING SAS VISUAL ANALYTICS REPORTS

Below is an example of listing the reports created withSAS Visual Analytics

```
async function example (store, logonPayload ) {
    let msg  = await store.logon(logonPayload);
    let root = await store.addServices('reports');

    let reports = await store.apiCall(root.links('reports'));
    printList( reports.itemsList() );
    let next;
    // do this loop while the service returns the next link
    while(((next = reports.scrollCmds('next')) !== null) )  {
        reports = await store.apiCall(next);
        printList( reports.itemsList() );
    }

    return 'All Done';
}

const printList =  (itemsList) => console.log(JSON.stringify(itemsList,
null, 4));

example(store, payload )
   .then (status => console.log(status))
   .catch(err => console.log(err));
```

REPEATING PATTERNS

One of the great strengths of restAF is to reduce your program logic to a few simple repeating patterns. The example above lists the reports. We will use the same logic to list a files collection. In the program below notice we just replaced “reports” with “files”.

```
async function example (store, logonPayload ) {
    let msg  = await store.logon(logonPayload);
    let root = await store.addServices('files');

    let files = await store.apiCall(root.links('files'));
    printList( files.itemsList() );
    let next;
    // do this loop while the service returns the next link
    while(((next = files.scrollCmds('next')) !== null) )  {
        files = await store.apiCall(next);
        printList( files,itemsList() );
    }

    return 'All Done';
}

const printList =  (itemsList) => console.log(JSON.stringify(itemsList, null, 4));

example(store, payload )
   .then (status => console.log(status))
   .catch(err => console.log(err));
```

The pattern we observed above lets us write a function that can be used to list any collection by adding the service name as an argument to the example function.

```
async function example (store, logonPayload, serviceName ) {
    let msg  = await store.logon(logonPayload);
    let root = await store.addServices(serviceName);

    let files = await store.apiCall(root.links(serviceName));
    printList( files.itemsList() );
    let next;
    // do this loop while the service returns the next link
    while(((next = files.scrollCmds('next')) !== null) )  {
        files = await store.apiCall(next);
        printList( files,itemsList() );
    }

    return 'All Done';
}

const printList =  (itemsList) => console.log(JSON.stringify(itemsList, null, 4));

example(store, payload )
   .then (status => console.log(status))
   .catch(err => console.log(err));
```

See the sample repository listed below for more examples.

REPOSITORIES

restAF and with all the sample code are available on github.com with Apache 2 licensing model.

- https://github.com/sassoftware/restaf - This is the main library.
- https://github.com/sassoftware/restaf-uidemos – This repository has many examples of using restAF in web applications.
- https://github.com/sassoftware/restaf-apiexplorer – This is an application to explore all the SAS REST APIs.

These examples depend on two other repositories that are also available to you.

- https://github.com/sassoftware/restaf-uicomponents – This is a collection of react components used in the examples.
- https://github.com/sassoftware/restaf-server - A web server used by the demos. You can use this for your own applications.

CONCLUSION

It is hard to explain the full power of restAF in a short paper, so please read through the full documentation and try the demos for more detailed information. Once you go thru a few examples you will become efficient in writing either web applications or nodejs applications with restAF.

The introduction stated

"restAF takes advantage of the consistency of the REST APIs to reduce the responses from the server into a standard form, and it also manages the data on the client side. This reduction enables developers to interact with the server in a repeating pattern regardless of which service the application is using."

And

"With restAF, you focus on your application and not on the nitty-gritty details of setting up HTTP calls, processing the returned results and HTTP codes, parsing the returned payload, managing the data, and so on."

In the examples we did not refer to uri, http methods or the raw responses from an http call. The examples did not parse the raw responses, do special handling for pagination and do other boring and repetitive work. All that was done for you by restAF. The examples also demonstrated how to take advantage of the repeating patterns to write more efficient reusable code.

OPEN SOURCE LIBRARIES

restAF uses three great open source projects(among others).

1. redux-saga (https://redux-saga.js.org/)
2. immutable-js (https://facebook.github.io/immutable-js/docs/#/)
3. axios (https://github.com/axios/axios)

The demo applications use more great open source projects:

1. react (https://reactjs.org/) and many open source react components

2. material-ui (http://www.material-ui.com/#/) for some of the UI components
3. hapi (https://hapijs.com/) for the restaf-server

ACKNOWLEDGEMENTS

My sincere thanks to the developers of the SAS Viya REST API for their patience in answering my many questions about their API.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Deva Kumaraswamy
100 SAS Campus Drive
Cary, NC 27513
SAS Institute Inc. deva.kumar@sas.com
http://www.sas.com

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.