Paper 4678-2020

How to Make Your First Impressive Web Application with Stored Processes and a Web Browser

Philip Mason, Wood Street Consultants Ltd.

ABSTRACT

Many organizations are experiencing the value of applications that can be run from a web browser. We all know that virtually anything is possible using this technology, which will usually consist of some HTML, CSS, and JavaScript running in the browser, with various other software running on a server. SAS® has provided the Stored Process Web Application that lets us connect the web browser to the SAS server, which opens up an enormous range of potential applications. In the simplest form, we can use a stored process to prompt the user for some info, run SAS code using that info, and return results to the web browser. From this simple starting point, this paper shows you how to make a more powerful and flexible web application. The emphasis is on showing those who know SAS (and nothing about web technologies) as simply as possible the steps to build a generic web application that they can use to start building their own web applications.

INTRODUCTION

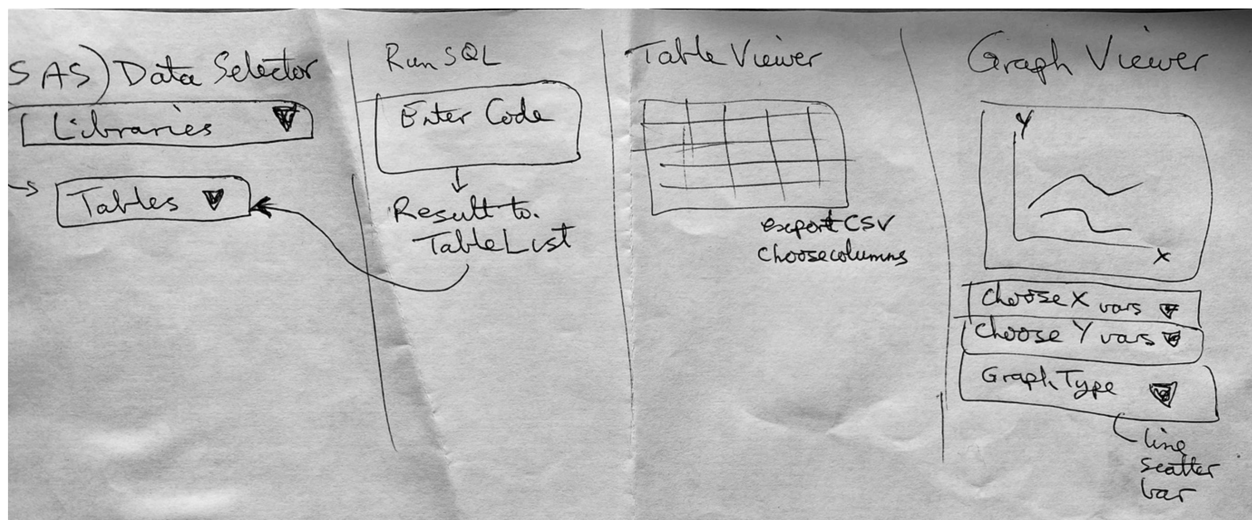
I have been writing papers and a book around the subject of using Stored Processes in various ways for many years now. A macro can take a SAS program and provide a way to generalize it, automate it, make it more flexible, usable in many more situations, and so on. In the same way a Stored Process raises the capability of any SAS program, including those already making use of macros. It allows you SAS program to be called not only by other SAS code but by other programming languages and to deliver output in a wide range of ways using SAS capabilities like ODS but also extending to using many other kinds of output.

One of the best ways to make use of SAS Stored Processes is to link your SAS system in with a web application. We all know that web applications are hugely flexible and powerful these days. There are frameworks and libraries widely used such as Angular, React, Bootstrap, GIT, etc. etc. Some of these you might have heard of, and I will be covering an example making use of some of these. Using a framework and various libraries gives you a way to leverage the capabilities created by others so you can get something quite powerful put together relatively quickly. Let's see what you think.

I'll try to keep things as simple as possible for the non-web programmer and also keep the SAS code as simple as possible.

MAKE A PLAN

First thing we should do is work out what kind of web application we would like to create. You don't need any special skills to do this so I'll grab a piece of paper and sketch out the web application that I would like to create. Then I will start going through the process to try to create this with application and show you how it's done.



So that is what I will try to create. This application will let you:

- select data from SAS
- run SQL to produce new data
- view tables
    - export a table as CSV data
    - choose what columns are displayed in table
- view graphs based on selected data
    - choose the x & y variables
    - choose from a range of graph types

GET YOUR TOOLS READY

Now that we have a target to aim for we need to prepare to start coding. It's important to choose good tools to help you in your project since that can help in all kinds of ways.

PICK AN EDITOR

In looking for an editor you should look for one with some of these features:

- understands different languages and will highlight code appropriately, auto-complete, auto-format code, identify non-matching things, etc.
- has integrated version control, preferably supporting GIT which is the most popular version control software.
- Has a really good editor with lots of useful shortcuts, multiple programs able to be open concurrently, etc.
- Automatically can recognize changes in code have been made and prepare it for testing
- Can manage other tools needed in your environment, such as web server software. e.g. can start your web server
- Support debugging of various kinds of code
- Integrated help

- 3rd party add-ons, which can provide lots of functionality and also indicate that it is something good which others have bothered to write addons for
- Integration with npm or other library managers, which make it easy to choose libraries to use and have them loaded automatically

The best tool I have found for doing development has been IntelliJIDEA which does have a free community version as well as more fully featured paid-for versions. You could also go with something like NotePad++ or VSCode. This tool is where we will be able to write almost all our non-SAS code.

The web application that we create is going to be made up of various kinds of web code for the front end as well as SAS code that will run in the back-end. The SAS code will provide data that the front end will consume. The front-end and back-end will be connected by the SAS Stored Process Web Application, which is a web app provided by SAS which allows us to run Stored Processes in a SAS environment while returning output to a web environment.

For my example I am going to use IDEA.

PICK A FRAMEWORK

This is not strictly necessary as you could just start writing HTML and JavaScript to create your own web application. However, you can gain a lot by choosing a good framework to develop within. React is the one I am going to use, as it's the best one I have found so far and I have been using most recently. React is created and maintained by Facebook and it is what Facebook uses. It's a bit technical for the beginner to understand why react is better than some other method but trust me and skip the explanation if you don't understand. Here are some of its great features:

- Virtual DOM – the Domain Object Model is manipulated in a web application to make it function. However normal DOM manipulation is slow. The virtual DOM is basically a way to keep a lightweight copy of the DOM and control its manipulation which results in huge speed gains.
- One-way data flow – data flows from a parent to child, and cant flow back up which gives greater control. Read-only properties are used for this, but a child can communicate with a parent using callback functions.
- Components – web pages are made up of components which define a view or part of a view. Logic is written in JavaScript and data can be passed around through the app keeping it our of the DOM.
- JSX – an extension to JavaScript which is similar to HTML. Components are written in JSX and it is a bit like a combination of HTML and XML. It makes producing components and using them very easy. React uses Babel to convert JSX to JavaScript. This means that you can use all the newest features of JavaScript in ES6 in your code, which is a great advantage. e.g.

```
const elem = <h1>Sales Report</h1>
```

- Conditional statements – these can be used in JSX and make it much easier to put logic into your application, e.g. if I have a value for a table variable then show me a table.
- Lifecycle methods – components in React have a particular lifecycle and you can execute code at different points in the lifecycle using methods such as shouldComponentUpdate, componentDidMount, componentWillUnmount, componentDidUpdate, etc.

The main alternative to react would be Angular. React is made by Facebook but Angular is made by Google.

GET A PACKAGE MANAGER

A package manager is a tool that will automate the management of packages on your system! It will install, update, configure and remove various packages that you might want to use in your development. It's a cornerstone of modern web development and highly recommended. The one that I use and recommend is NPM which stands for Node Package Manager and comes with Node.js. It has access to the world's largest software library with access to over 800k packages. You can download an installer for your system and make this available to use.[1]

One great feature of npm is that it manages dependencies for you. So if you want a graph package which needs 3 other packages installed in order for it to work, then npm will install everything in the right way so that it works for you.

Once you have npm installed you will be able to install modules using commands like this:

```
npm install underscore
```

CREATE A STARTER REACT WEB APPLICATION

npx is a tool that comes with npm when you install it. It is an npm package runner and will download a package and then run it. We are going to use an npx command to create a single page web application example. This is provided by react for programmers so they can build their application from it as a starting point. We simply issue the following command to get our React web app setup (I called it data-viewer but you can name it whatever you like).

```
npx create-react-app data-viewer
```

Here is the first part of what I see when I run this from Terminal in IDEA. The process goes on for some minutes until everything is installed and configured.

```
Philips-MacBook-Pro-2:sgf philipmason$ npx create-react-app data-viewer

Creating a new React app in /Users/philipmason/IdeaProjects/sgf/data-viewer.

Installing packages. This might take a couple of minutes.
Installing react, react-dom, and react-scripts with cra-template...

yarn add v1.15.2
[1/4] Resolving packages...
[2/4] Fetching packages...
[3/4] Linking dependencies...
warning "react-scripts > @typescript-eslint/eslint-plugin > tsutils@3.17.1" has unmet peer dependency "typescript@>=2.8.
dev || >= 3.7.0-beta".
warning "react-scripts > eslint-config-react-app@5.2.0" has incorrect peer dependency "eslint-plugin-flowtype@3.x".
```

After it all finishes I see this.

```
Success! Created data-viewer at /Users/philipmason/IdeaProjects/sgf/data-viewer
Inside that directory, you can run several commands:

  yarn start
    Starts the development server.

  yarn build
    Bundles the app into static files for production.

  yarn test
    Starts the test runner.

  yarn eject
    Removes this tool and copies build dependencies, configuration files
    and scripts into the app directory. If you do this, you can't go back!

We suggest that you begin by typing:

  cd data-viewer
  yarn start
```

Now I can go and issue the commands suggested to see the React web application.

[1] https://nodejs.org/en/download/

MAKE THE BONES OF YOUR WEB APPLICATION

We can now go in and edit app.js (as it says in the web app). That is a JavaScript program which displays the page we see. We will modify this to have some basic placeholders for various things we want to appear in our web app.

```
import React from 'react';
import './App.css';

function App() {
  return (
    <div className="App">
      <h1>Select SAS Data</h1>
      <h2>Library Selector</h2>
      <h2>Table Selector</h2>
      <h1>Run some SQL</h1>
      <hr/>
      <h1>Table Viewer</h1>
      <h1>Graph Viewer</h1>
    </div>
  );
}

export default App;
```

Select SAS Data

Library Selector

Table Selector

Run some SQL

Table Viewer

Graph Viewer

Now we can go through each of these placeholders and replace them with working code to do what we want. Now we need to work out how we are going to implement each of these things. Choosing how to do this requires some experience, advice from others or exploring the various libraries available on npm especially looking for ones that work well with React. Some libraries are written specifically to work nicely with React, but you can usually make things work with it. For instance we use a jQuery library called DataTables with React because it is so good, even though there are other simpler options.

SELECT SOME COMPONENTS

Let's think about what we need:

- Drop down menu for selecting libraries and tables.
- Text entry area so SQL code can be typed in.
- Something that will let us view data as a table.
- Something that will let us view data as a graph.

There are many packages of code that can be used for these things and we could do a search for something like "react drop down menu" then searching through the entries to find something nice to use. You can see a list of great packages here - https://github.com/brillout/awesome-react-components

Having looked for good but simple packages to do what we need I suggest using:

- Drop down menus are handled well by React Select - https://react-select.com/home - install using this command:

```
npm install react-select
```

- Text areas are handled by standard HTML but we can get some advantages using React Bootstrap - https://react-bootstrap.github.io/components/forms/ This will also help us layout our page nicely later. Install using this command:

```
npm install react-bootstrap bootstrap
```

- Tables can be viewed using the React Data Table Component - https://adazzle.github.io/react-data-grid/ or https://jbetancur.github.io/react-data-table-component/?path=/story/general--kitchen-sink - install using this command:

```
npm install react-data-table-component styled-components
```

- Graphs can be viewed using React Google Charts - https://github.com/RakanNimer/react-google-charts - install using this command:

```
npm i -s react-google-charts
```

SWITCH TO USING COMPONENTS

Now we can gradually replace our place holder HTML with some real components. We will make some test data so we can just get each component up and running and see what they look and work like. After that we can change our test data over to using data supplied by SAS.

React Select

There are just a few things we need to do in order to make use of this component.

1. Add a line in the code to import the package. This loads it in ready for use. In react things will just be loaded when needed.

```
import Select from "react-select";
```

2. Add a few lines to define some test data. For a select (drop down menu) we need data that specifies a value and a label (there is more we could specify such as disabled).

```
const sasLibraries=[
    {value:"a", label:"Lib A"},
    {value:"b", label:"Lib B"}
];
const sasTables=[
    {value:"a.tab1", label:"Table 1"},
    {value:"b.tab2", label:"Table 2"}
];
```

3. Replace our placeholder <h1> with a <Select> tag that uses the test data.

```
<Select options={sasLibraries} />
<Select options={sasTables} />
```

This produces this select menu. If we make a selection, we will be able to pick up the value selected in JavaScript

React Bootstrap

There are just a few things needed to make use of React bootstrap. There are lots of components within bootstrap, so once you do the general things, you need to import whichever bits you want to use. Its' good to do this so you only ever send the required code down to the client.

1. You need to include some CSS which define Style Sheets to be used, since some are required. This can be done like this.

```
import 'bootstrap/dist/css/bootstrap.min.css';
```

2. Next we need to import whichever components of React Bootstrap that we are going to use. So I am going to use a Row and Column for laying out my page, since bootstrap is really good at that. The following lines will import the code needed for that.

```
import Row from 'react-bootstrap/Row';
import Col from 'react-bootstrap/Col';
```

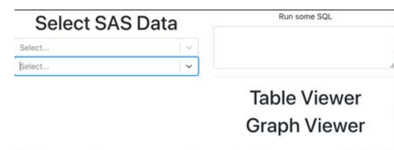
3. Then I can wrap code with tags: <Row> <Col> </Col> </Row>. Additionally there are properties that can be specified with the tags and I will use md="6" on the Col tag. Bootstrap divides the screen up into 12 parts across the screen, so 6 means to use half the screen for that column on medium sized screens (which is the md). I wont show all the code now, but we will see it all at the end of this section.

4. I'm also going to use React Bootstrap to put in a text area where I can enter some SQL code to run. I need to import the Form code like this:

```
import Form from 'react-bootstrap/Form';
```

5. And then I can put the text area in using this code:

```
<Form.Label>Run some SQL</Form.Label>
<Form.Control as="textarea" rows="3" />
```



6. So now we have laid out of screen in 2 columns, we have 2 drop down menus and an area to enter some SQL into.

React Data Table Component

To use this we import the package, make some data and then use the component. So the steps are:

1. Import the package.

```
import DataTable from 'react-data-table-component';
```

2. We create an array of objects to describe the columns of the table. And then we create another array of objects, each of which is a row of data. e.g.

```
const columns = [
    { selector: 'id', name: 'ID' },
    { selector: 'name', name: 'Name' },
    { selector: 'amount', name: 'Amount' } ];
const data = [
    {id: 0, name: 'Phil', amount: 20},
    {id: 1, name: 'Jake', amount: 40},
    {id: 2, name: 'Annie', amount: 60} ];
```

3. Finally we use the component and pass some property values to specify the data and column definitions.

```
<DataTable
    title="Table Viewer"
    columns={columns}
    data={data}
/>
```

Table Viewer

ID	Name	Amount
0	Phil	20
1	Jake	40
2	Annie	60

This will produce a nice table for us, with many other properties we can use to customize it more.

React Google Charts

To use React Google Charts we need to do the same as the other packages we have used above. We import the package, prepare some data in the required form and then use it. So the steps are:

1. Import the package

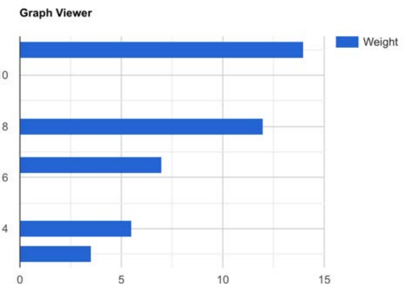
```
import { Chart } from "react-google-charts";
```

2. Make some data we can plot. The data for google charts can be provided in an array, where the first row has labels to use and then rows follow with the data.

```
const dataForGraph = [ ["Age", "Weight"],
    [8, 12], [4, 5.5], [11, 14], [4, 5], [3, 3.5], [6.5, 7]  ];
```

3. Finally we use the package and data together to make a graph.

```
<Chart
    chartType="BarChart"
    data={dataForGraph}
    width="80%"
    height="400px"
    legendToggle
/>
```



With each of these packages we have installed we are using them in the simplest way. There are lots of features and power built into these packages. Later in the build process we will need to use some more features, such as running SAS code when we make a selection from a drop-down menu.

NEXT STEPS

Now that we have a basic web application with some dummy data, we need to populate the components with proper data from SAS and then link things together, so they do something. So the steps we will follow are:

1. Create SAS Stored Processes that take parameters and return JSON data.
2. Call the Stored Processes using JavaScript. That will enable us to take selections made from drop down menus or SQL code entered and send to the stored processes as parameters. The SAS code can use that data, execute and then return results in JSON form. The JSON data returned is automatically converted into JavaScript objects which we can make use of.
3. The data returned from stored processes will be fed to the various components to use, such as the drop-down menus, graph and table. To make this happen we need to cause functions to run when we make selections. This is typically handled by handlers that will execute something when something is clicked on or typed in.

MAKE THE SAS STORED PROCESSES

WHAT STORED PROCESS ARE NEEDED?

Let's work out what stored processes we will need. Wherever we need to get some data from SAS to populate something in the web application, we will need a stored process to call.

1. get_libs – will be a stored process that will simply return SAS libraries, which can then be displayed in the library selection drop down menu.
2. get_tables – will be a stored process that will return tables that belong to a SAS library. The user will make a selection of a library and that will be passed in as a parameter. Table names will then be returned that can be used to populate the table selection drop down menu.
3. get_table_data – will be a stored process that will take a library and table name as parameters (perhaps combined as lib.table) and return the data for that table. The

data will be in a form that can easily be used for the table component that we are using.

4. get_graph_data – this stored process takes a table as input and returns the data for it. The data needs to be in a form that is easily consumable by our graph. We have separate stored processes for the table and graph since the formats differ a bit and it is simpler to do it this way. Though it would be possible to just use one stored process and then to manipulate the data in JavaScript.
5. run_sql – will be a stored process that we use to run some SQL code in SAS and return data that can be used with the graph and table. We will expect the SQL to just be a select statement without a create table in front of it. In that way we can add a "create table temp as" in front of the SQL so that we know what table is actually created. Then we can return the data from that table.

WHAT SHOULD BE OUTPUT?

It is useful to work out what kind of data we will need to be returned. We can get a good idea of that from the example code we have so far where we have made some test data to populate the objects. You can also go and look at the documentation for each package we are using to see what kind of data is needed by them.

In JSON arrays are surrounded by [ ]. Elements in an array are separated by commas. e.g.

```
[“red”, “green”, “blue”]
```

Objects are surrounded by { }. Elements in an object consist of a key and a value, and you can have as many as you want. e.g.

```
{name: “Phil”, age: 55}
```

All of the data we are writing out from our stored processes will be in JSON form. This is really key to the way that we can best create web applications with SAS and JavaScript. SAS gets the data and provides it to JavaScript which then makes use of it. In that way we use both for their strengths.

1. get_libs – React-Select which makes our drop down menus basically needs a value and label for each line that is displayed. So our stored process needs to get data required and then write it out in JSON form that has value and label in objects that are easch items in an array. We will use SQL to access the dictionary table for libraries to get a list of them. Then we can use PROC JSON to write that data out in the required format.
2. get_tables – This stored process will take a SAS library name as an input parameter. We will use SQL to access the dictionary table for tables to get a list of them matching the library that was specified. Then we can use PROC JSON to write that data out in the required format for React-Select.
3. get_table_data – This will be getting data for the specified table, which is passed as an input parameter. The data needs to be output in a form that can be used by our table.
4. get_graph_data – Same as the previous one, but data needs to be easily usable by graph. This data will be an array made up of arrays. The first array contains labels for the x and y variables. Other arrays will each have an x and y value.
5. run_sql – This stored process will take some SQL code as a text string as input. That SQL code will be run and the data returned in a form that can be shown in the table. In fact we could include code from get_table_data to do this, or use a macro to share the code between the stored processes. Or from a JavaScript perspective we could just have this stored process run the SQL and then use the other stored process to get the data.

There are always lots of options and also choices to be made about whether we do more processing in SAS before sending data to JavaScript or do processing in JavaScript and therefore not need as much to be done in SAS.

RESTRUCTURING OUR CODE

It is worth restructuring our code from what was provided by the sample React starter app we based things on. We want to make use of one particular useful feature of react, which is state management. For that we will convert out App function to a Component. It is quite simply done and lets us do things like run some code when a component starts or updates. And we can set properties for the state of a component and then have it render again when values of those components change. We will make use of this in particular so that when a user chooses a library we can run a stored process to get the tables in that library, put that data into the state and then the component will be updated with them.

Some of the key bits of code that we will now have include:

```
class App extends React.Component {
    constructor(props) {
        // this is so we could pass parameters in - good practice for
components
        super(props);
        this.state={} ; // initialise the state or we get an error
```

We change from a function to a class, with the same name of App.

We add a contructor section which is run before the component is displayed, so we can get things ready like loading data.

We add in super(props) which allows us to pass in parameters to this class. It's good practice to add this in even if we aren't yet passing parameters in.

Then we initialize this.state which will hold that state of various properties in our component. We can add the list of libraries/tables/variables/data and so on to this.state.

We also need to add in a render function, which will be called whenever we need to render the screen again, like when something changes in the state. The render will start like this:

```
render() {
    return (<div className="App">
        <Row>
```

We will change the data being passed in to various components by using it from the state rather than just from a variable. So it will look like this:

```
<Select
    options={this.state.sasLibraries}
/>
```

Finally, at the end we will export the class using some code like this:

```
export default App ;
```

At this point we haven't used any SAS code, so it could all be developed on any computer with or without SAS. The code is listed in the section at the end showing Source Code.

CODE FOR STORED PROCESSES

Now we are going to create stored processes to take parameters and return JSON data to feed all the components we are using. You can make stored processes and copy the code in so you can test these in your own environment. All you need to do is also define any parameters mentioned as well as pasting the code in, and using other default settings. If you are unsure of any of this you can look back at my previous papers from the last 10 years where I have covered this many times.

get_libs

We don't need any parameters for this one, since it will just return JSON data for use with react-select with all the libraries available. Here is the SAS code for the stored process.

```
proc sql;
  create table info as select distinct libname as value,
libname as label from dictionary.tables where libname ne "WORK";
quit;

proc json out=_webout nosastags pretty;
  export info;
run;
```

get_tables

We will specify a library and then get a JSON data for use with react-select with all the tables for that library. So we define a parameter called *libname* in the stored process. Here is the SAS code for the stored process.

```
proc sql;
  create table info as
    select distinct strip(libname)||'.'||memname as value, memname as label
      from dictionary.tables where libname = "&libname";
quit;

proc json out=_webout nosastags pretty;
  export info;
run;
```

get_variables

We will specify a parameter called *memname*, which will have the libname and table name separated by a dot – in the standard SAS fully specified dataset name way. E.g memname=sashelp.baseball . Here is the SAS code for the stored process.

```
proc sql;
  create table info as
    select distinct name as value, strip(label)||'
('||strip(name)||')' as label
      from dictionary.columns where strip(libname)||'.'||memname
= "&memname";
quit;

proc json out=_webout nosastags pretty;
  export info;
run;
```

get_table_data

We will pass in the name of the table we want as a fully specified dataset name with libname and table name. e.g. memname=sashelp.class. Here is the SAS code for the stored process. We will also specify a parameter called maxobs which will have a default value of 1000. This will be the maximum number of records we want to bring back as data. Having a maximum means we cant accidentally get stuck with a very long task of perhaps bringing millions of records back, unless we override the default value. Here is the SAS code for the stored process.

```
proc sql;
  create table data as
    select *
```

```
      from &memname(obs=&maxobs);
    create table columns as
      select distinct name as selector, name
        from dictionary.columns where strip(libname)||'.'||memname
  = "&memname";
  quit;

  proc json out=_webout nosastags pretty;
    write open object;
    write values "columns";
    write open array;
    export columns;
    write close;
    write values "data";
    write open array;
    export data;
    write close;
    write close;
  run;
```

get_graph_data

For this we will specify the table name which we will call memname (e.g. sashelp.class), x-axis variable to use which we will call xAxis (e.g. name) and y-axis variable to use which we will call yAxis (e.g. weight). We will also have a parameter called maxobs which will be the maximum number of records to bring back and have a default of 1000, to avoid bringing back too many records unexpectedly. One thing to notice in this stored process is that we wont make use of PROC JSON since we want some specially laid out JSON returned. This is very easily achieved with a simple data step to read the data in and write JSON out.

```
   data _null_;
    file _webout;

    if _n_=1 then
      put '[' / "['&xAxis','&yAxis'],";
    set &memname end=_end;

    if vtype(&xaxis)="C" then
      _&xaxis=quote(strip(&xaxis));
    else _&xaxis=&xaxis;
    put '[' _&xaxis ',' &yaxis ']' @;

    if _end then
      put ']';
    else put ',';
  run;
```

run_sql

This stored process will take some SQL code as an input parameter and return JSON with the data produced from running the SQL. So we will make a parameter called sqlCode that will have the SQL to run. e.g. "select * from sashelp.class". Here is the code for the stored process.

```
  proc sql;
    create table data as
      %superq(sqlCode);
    create table columns as
      select distinct name as selector, name
        from dictionary.columns where libname = "WORK" & memname="INFO";
```

```
quit;

proc json out=_webout nosastags pretty;
  write open object;
  write values "columns";
  write open array;
  export columns;
  write close;
  write values "data";
  write open array;
  export data;
  write close;
  write close;
run;
```

INTEGRATE THE STORED PROCESSES INTO JAVASCRIPT

HOW TO CALL A STORED PROCESS

In JavaScript there are a few ways we can makes calls to a URL and capture the output returned. We are going to use what I think is the best method within React. We can use a fetch function to make an AJAX call to run the stored process via the SAS Stored Process Web Application. That returns a Promise which indicates if it worked or not and allows us to pick up the data returned. This is all explained at this web page - https://JavaScript.info/fetch. The nice thing about things that use promises is that we can make a bunch of asynchronous calls and then define what happens when each of them finish. We can even use await to pause in execution of JavaScript and wait for an asynchronous process to complete – effectively making it synchronous.

Here is an example of calling a URL which returns some JSON, like a SAS Stored Process would. Here we are calling reqres.in which allows us to pass some JSON in and get a response which returns that JSON. This is great for testing purposes and means that if you don't have SAS access on your computer you can still develop the web application without live data by just passing in some test data you will get back. It is then easy to change over to your SAS Stored Process web application later.

The other things to note with this JavaScript code is that we are using a promise chain, which means it will do the fetch, and then if that works does the *.then* bit after it. Then it that works it does the next one, and so on. You can also specify a *.catch* to handle any errors if you like.

```
fetch("https://reqres.in/api/users",
    {
        method: "post",
        headers: {'Content-Type': 'application/json'},
        body: JSON.stringify([
            {value: "a", label: "Lib A"},
            {value: "b", label: "Lib B"}])
    })
    // convert the text JSON returned to a   object
    .then(response => response.json())
    // putting data in state means that when it changes our page will re-render
    .then(data => {
        this.setState({sasLibraries: data});
        console.log("sasLibraries", data);
    });
```

Here is an example of a SAS Stored Process being called, and then the response being converted to an object and that object put into the state. This is exactly what we need to do with each fetch to a stored process.

```
fetch("http://myserver/SASStoredProcess/do?_program=/User+Folders/phil/Data+Viewer/get_libs")
    .then(response => response.json
        .then(data => this.setState( { sasLibraries: data } );
```

Here is another example of a SAS Stored Process being called but in this case we are also passing a parameter to the stored process. The parameter is added to the end of the URL in the fetch by having an ampersand followed by the parameter name and then the value. In this example it is equivalent to doing a macro assignment in the stored process like %let libname=SASHELP;

```
fetch("http://myserver/SASStoredProcess/do?_program=/User+Folders/phil/Data+Viewer/get_libs&libname=SASHELP")
    .then(response => response.json
        .then(data => this.setState( { sasLibraries: data } );
```

LINKING ACTIONS TO STORED PROCESS CALLS

Let's think about how a user would use this application. They would select a library, then a table and the table would then appear on screen. Before a graph can appear, we would need to choose an x & y variable, so we need to prompt the user for that information. Then a user additionally could enter some SQL into the text area which could be run to produce data and bring it back to show on screen. We really need to add a button so that the user can enter their SQL and then press "run" to execute it. So, we have identified several things we need to add to the user interface – button to run the SQL and menus to select x variable and y variable.

For the button we can use a button component from React-bootstrap. We can add a button by importing the button component from bootstrap-select[2] and using it[3].

And for the x & y variables we can again use react-select by giving the component a list of variables in the table[4] from which the user can select one. Of course, you also need a bit of code to make a menu for the x-axis variable selection, and one for the y-axis[5]. This will actually need another stored process that we can call to get a list of variables in a table.

The actions needed to make all this happen would be:

- Load list of libraries when app starts by running get_libs. This can be done as the web app starts and doesn't need to be done again when the user makes a selection or anything.
- When a library is picked load list of tables by running getTables. We can add an onChange property to call a function when a change is made in the library menu. The function will run the stored process and pass the data to the tables menu.

```
<Select
    options={this.state.sasLibraries}
    onChange={this.getTables}
/>
```

- The function getTables will run the stored process get_table_data to get us a list of tables in the libname that was selected

---

[2] import Button from 'react-bootstrap/Button';

[3] <Button variant={"primary"} block>Run SQL</Button>

[4]
```
const sasVariables=[
   {value:"var1", label:"Variable 1"},
   {value:"var2", label:"Variable 2"}
];
```

[5]
```
<Select options={sasVariables} placeholder={"Choose x-axis variable"}/>
<Select options={sasVariables} placeholder={"Choose y-axis variable"}/>
```

```
getTables = (e) => {
    console.log("e.value",e.value);

fetch("https://myserver/SASStoredProcess/users/Phil+Mason/get_table_data&libname="+
e.value)
        .then(response => response.json())
        .then(data => this.setState({sasTables: data } ) );
}
```

- When a table is picked load a list of variables that are contained in that table by running get_table_data. That stored process will also run get the stored process to get the data for displaying in tabular form. Again, we will use the onChange property to run a function when the user picks something in the list of tables.

```
<Select
    options={this.state.sasTables}
    onChange={this.getVariables}
/>
```

- The function getVariables will run the stored process get_variables that will return a list of SAS variables that can be used for selecting x and y axis variables for the graph.

```
getVariables = (e) => {
    console.log("e.value",e.value);

fetch("https://myserver/SASStoredProcess/users/Phil+Mason/get_variables&memname="+e
.value)
        .then(response => response.json())
        .then(data => this.setState({sasVariables: data } ) );
    this.getData(e) ;
    // put the selected table into state so we can use it elsewhere too
    this.setState({selectedTable: e.value});
}
```

- the getData function will call the stored process get_data_columns and get_data, which are needed to provide the data for displaying a grid of data.

```
getData = (e) => {
    console.log("e.value",e.value);
    // get column spec of table

fetch("https://myserver/SASStoredProcess/users/Phil+Mason/get_data_columns&memname=
"+e.value)
        .then(response => response.json())
        .then(data => this.setState({columns: data } ) );
    // get data of table

fetch("https://myserver/SASStoredProcess/users/Phil+Mason/get_data&memname="+e.valu
e)
        .then(response => response.json())
        .then(data => this.setState({data: data } ) ) ;
}
```

- When we have a value entered for the x & y variables, they will run a function either called setX or setY.

```
<Select options={this.state.sasVariables} placeholder={"Choose x-axis variable"}
onChange={this.setX}/>
<Select options={this.state.sasVariables} placeholder={"Choose y-axis variable"}
onChange={this.setY}/>
```

- setX and setY will set the value of the variable selected into state, so it can be used elsewhere. Then it will get to see if both x and y have been specified. If they have then it will run the function getGraphData to get the data needed for the graph.

```
setX = (e) => {
    this.setState({xAxis: e.value});
```

```
    if (this.state.hasOwnProperty("yAxis")) this.getGraphData() ;  // if we have x 
& y specified then get graph data
}

setY = (e) => {
    this.setState({yAxis: e.value});
    if (this.state.hasOwnProperty("xAxis")) this.getGraphData() ;  // if we have x 
& y specified then get graph data
}
```

- getGraphData will run the stored process get_graph_data, passing in the table name, x and y axis variable names. It will then get back data needed to make the graph and assign that to a state variable.

```
getGraphData = () => {

fetch("https://myserver/SASStoredProcess/users/Phil+Mason/get_graph_data&memname="+
        this.state.selectedTable+
        "&xAxis="+this.state.xAxis+
        "&yAxis="+this.state.yAxis)
        .then(response => response.json())
        .then(data => this.setState({dataForGraph: data } ) ) ;
}
```

- When SQL has been entered and a pushbutton is pressed then we run call the function runSql.

```
<Form.Label>Run some SQL</Form.Label>
<Form.Control as="textarea" rows="3"
              ref={this.sqlRef}/>
<Button variant={"primary"} block onClick={this.runSql}>Run SQL</Button>
```

- The function runSql run the SQL code in SAS using the run_sql stored process and then download the column definition and data needed to display it in the table. By updating the variables in state will make the new data display in the table on screen.

```
runSql = () => {
    const sqlCode=this.sqlRef.current.value;
    console.log("sqlCode",sqlCode);

fetch("https://myserver/SASStoredProcess/users/Phil+Mason/run_sql&sqlcode="+sqlCode
)
        .then(response => response.json())
        .then(data => this.setState({
            columns: data.columns,
            data: data.data
        } ) ) ;
}
```

SUMMARY TABLE

A nice way to summarize what is going on in the JavaScript program is to use a table. We can see what JavaScript function is used, the React Component, the data used for the component from this.state, data provided to that component, action called (by selecting something or clicking), stored process called, data returned from the stored process and loaded into JavaScript, the state variable set with the info we need, any additional function also called.

function	*Component*	*state data source*	*action*	*stored process*	*data returned*	*sets state*	*also calls*
constructor				get_libs	list of libnames available	sasLibraries	
render	Select	sasLibraries	getTables				
getTables				Get_tables	list of tables in a library	sasTables	

render	Select	sasTables	getVariables				getData
getVariables				get_variables	list of variables in table	sasVariables	
getData				Get_table_data	columns spec & data	columns	
	Form.Control	this.sqlRef[6]					
	Button		runSql				
runSql				run_sql	columns spec & data	columns, data	
	DataTable	columns, data					
	Select	sasVariables	setX				
	Select	sasVariables	setY				
setX			getGraphData			xAxis	
setY			getGraphData			yAxis	
getGraphData				get_graph_data	Data needed for graph	dataForGraph	
	Chart	dataForGraph					

CONCLUSION

I hope that I have shown you how you can build up a web application from something quite simple and progressively add a bit more functionality without making it too complex. Of course you can do almost anything in JavaScript for a web application. There is a huge number of different component libraries out there to help you too. So I suggest imagining something great and start building it from a simple start.

I hope I have also shown that there can be quite a division in coding between SAS and web technologies. So if you have good SAS programmers, get them to learn how to make stored processes and deliver output in JSON form. Then they can provide great APIs to web programmers who can develop the web side of things. You can also easily mix in things like python and other languages in the backend which can provide JSON data to your web apps. Or you can link in your SAS systems to existing web applications which perhaps require feeds from SAS, since you can provide APIs that will directly feed data to those systems when it is needed. It's all very flexible really.

SOURCE CODE

Here is the full source code for the web application.

```
import React from 'react';
import './App.css';
import Select from "react-select";
import 'bootstrap/dist/css/bootstrap.min.css';
import Row from 'react-bootstrap/Row';
import Col from 'react-bootstrap/Col';
import Form from 'react-bootstrap/Form';
import Button from 'react-bootstrap/Button';
import DataTable from 'react-data-table-component';
import Chart from "react-google-charts";

class App extends React.Component {
    constructor(props) {
```

[6] Not really a state variable, but a reference we define so that we can pick up the value entered in the text box. Gives us a handle to use to address the data in the text box.

```
        super(props);   // this is so we could pass parameters in - good practice for
components
        this.sqlRef = React.createRef();
        this.state={
            sasTables: [],
            sasVariables: [],
        } ; // initialise the state or we get an error

        // call Stored Process to get a list of libraries
        fetch("https://myserver/SASStoredProcess/users/Phil+Mason/get_libs",
        )
            .then(response => response.json())
            .then(data => this.setState({ sasLibraries: data } ) ) ;
    }

    getTables = (e) => {
        console.log("e.value",e.value);

fetch("https://myserver/SASStoredProcess/users/Phil+Mason/get_table_data&libname="+e.v
alue)
            .then(response => response.json())
            .then(data => this.setState({sasTables: data } ) );
    }

    getVariables = (e) => {
        console.log("e.value",e.value);

fetch("https://myserver/SASStoredProcess/users/Phil+Mason/get_variables&memname="+e.va
lue)
            .then(response => response.json())
            .then(data => this.setState({sasVariables: data } ) );
        this.getData(e) ;
        // put the selected table into state so we can use it elsewhere too
        this.setState({selectedTable: e.value});
    }

    getData = (e) => {
        console.log("e.value",e.value);
        // get column spec of table

fetch("https://myserver/SASStoredProcess/users/Phil+Mason/get_data_columns&memname="+e
.value)
            .then(response => response.json())
            .then(data => this.setState({columns: data } ) );
        // get data of table

fetch("https://myserver/SASStoredProcess/users/Phil+Mason/get_data&memname="+e.value)
            .then(response => response.json())
            .then(data => this.setState({data: data } ) ) ;
    }

    setX = (e) => {
        this.setState({xAxis: e.value});
        if (this.state.hasOwnProperty("yAxis")) this.getGraphData() ;  // if we have x
& y specified then get graph data
    }

    setY = (e) => {
        this.setState({yAxis: e.value});
        if (this.state.hasOwnProperty("xAxis")) this.getGraphData() ;  // if we have x
& y specified then get graph data
    }
```

```
    getGraphData = () => {

fetch("https://myserver/SASStoredProcess/users/Phil+Mason/get_graph_data&memname="+
            this.state.selectedTable+
            "&xAxis="+this.state.xAxis+
            "&yAxis="+this.state.yAxis)
            .then(response => response.json())
            .then(data => this.setState({dataForGraph: data } ) ) ;
    }

    runSql = () => {
        const sqlCode=this.sqlRef.current.value;
        console.log("sqlCode",sqlCode);

fetch("https://myserver/SASStoredProcess/users/Phil+Mason/run_sql&sqlcode="+sqlCode)
            .then(response => response.json())
            .then(data => this.setState({
                columns: data.columns,
                data: data.data
            } ) ) ;
    }

    render() {
        return (<div className="App">
            <Row>
                <Col md="6">
                    <h1>Select SAS Data</h1>
                    <Select
                        options={this.state.sasLibraries}
                        onChange={this.getTables}
                    />
                    <Select
                        options={this.state.sasTables}
                        onChange={this.getVariables}
                    />
                </Col>
                <Col md="6">
                    <Form.Label>Run some SQL</Form.Label>
                    <Form.Control as="textarea" rows="3"
                                  ref={this.sqlRef}/>
                    <Button variant={"primary"} block onClick={this.runSql}>Run
SQL</Button>
                </Col>
            </Row>
            <Row>
                <Col md={6}>
                    <DataTable
                        title="Table Viewer"
                        columns={this.state.columns}
                        data={this.state.data}
                    />
                </Col>
                <Col md={6}>
                    <Select options={this.state.sasVariables} placeholder={"Choose x-
axis variable"} onChange={this.setX}/>
                    <Select options={this.state.sasVariables} placeholder={"Choose y-
axis variable"} onChange={this.setY}/>
                    {/* When we have values for the xAxis and yAxis then the Chart
component will be called. */}
                    {this.state.xAxis &&
                    this.state.yAxis &&
                    <Chart
                        chartType="BarChart"
```

```
                        data={this.state.dataForGraph}
                        width="80%"
                        height="400px"
                        options={{title: "Graph Viewer"}}
                    />}
                </Col>
            </Row>
        </div>)
    }
}

export default App ;
```

CONTACT INFORMATION

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

Philip Mason
Wood Street Consultants
phil@woodstreet.org.uk

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.