TECHNICAL PAPER

Power up your Pages: Modern Solution Extensions in SAS® Visual Investigator

Last update: November 2024

Contents

Relevant Products and Releases

- SAS® Visual Investigator
  - 10.8
  - SAS Visual Investigator on SAS® Viya® 4 (cadence releases)

Introduction

Purpose

This document is designed to give a broad overview of solution extensions for SAS Visual Investigator on SAS Viya 4. It covers their construction, use, and maintenance at a level that is helpful for users with some technical background (such as SAS consultants or SAS Visual Investigator administrators). After reading this paper, users should be able to create solution extension projects and work with them in SAS Visual Investigator. They will also have some knowledge about the inner workings, if they decide to learn more about Angular or other web development tools.

Scope

This document assumes some comfort with executing commands on your development computer. Any needed commands are given, except for fundamentals such as navigating folders. Since solution extensions are bits of webpage code, some familiarity with the basic blocks—HTML for structure, CSS or similar for styling, and JavaScript for everything else—is helpful for taking the content in this document and working it into projects to fit your needs. This document explains some fundamentals of the Angular framework in a later section, but it also assumes some willingness to explore Angular documentation. Otherwise, this document does not assume that readers are familiar with any software packages that are used in descriptions or examples.

This document assumes some familiarity with SAS Visual Investigator and its administration, especially the construction of pages and entities. For more information about any unfamiliar items, visit the SAS Help Center to get documentation for your version of SAS Visual Investigator. To get the most out of tie-ins to SAS Visual Investigator, SAS Viya, or even the internet at large, familiarity with accessing REST APIs via HTTP is also helpful.

What are solution extensions? Why use them?

Solution extensions are a way that is prescribed by SAS to add functionality to SAS Visual Investigator pages. They enable developers to build an implementation that fits unique customer needs. To list a few real examples, they are elements such as text boxes that validate user input, displays that fetch instantly up-to-date information from third-party sites, or check boxes that require user acknowledgement before the object can be saved. Essentially, if it can run in a web browser, it can be done with solution extensions. This versatility makes them incredibly powerful and turns many questions of “can it be done in SAS Visual Investigator?” into questions of “how many developer hours do we have?”

As a more technical answer, solution extensions are web components as defined by WHATWG. To paraphrase, this means they are completely encapsulated bits of webpage (even if they do not have an interface that is visible to the user) that contain structure, style, and functionality in one bundle. Starting with SAS Visual Investigator 10.8, the SAS recommended way to generate these web components is with Angular. Angular is a front-end web app framework that uses the TypeScript language, which is a superset of JavaScript. It does the heavy lifting so that developers can focus on specialized logic and design. It’s worth noting, though, that any method resulting in a web component could be used to create a solution extension. For experienced developers with a preferred front-end web ecosystem, experimenting and sharing on outlets such as communities.sas.com can be a great way to get connected, give feedback to SAS R&D, and help your fellow users get the most out of their SAS Visual Investigator instance.

Getting Started

Your Development Environment

Because solution extensions are web components, the best tools for the job are those that are geared toward Angular and front-end web development. We're not going to make a claim that one option is best, but we do recommend looking for an integrated development environment (IDE) that has code checking for web languages, a file browser, and built-in command-line access. Nice-to-have features include plug-ins for Angular and TypeScript, as well as customizable build tasks. We use Visual Studio Code.

With an IDE in place, the next step is installing Node.js and npm. Node is a JavaScript engine that runs ancillary scripts to configure and install the code that we write. npm is the Node package manager, which we'll use to install SAS tools for developing solution extensions. As of this writing, the version of Node recommended by SAS is Node.js 14 (which comes bundled with the appropriate version of npm). These tools can be installed from the Node.js website.

Scaffolding

Now, navigate to a folder where you want to create a solution extension project. The use of "project" here is to specify that the folder and upload bundle created in the following steps can contain an arbitrary number of controls, which are the individual units that you add to your SAS Visual Investigator pages at the end. To create your project, run:

```
npx @sassoftware/vi-solution-extension-create
```

You are prompted to enter a few items, such as the name of your project, the URL of the server to which you are deploying, and your credentials. For more details about this process, the readme file for the vi-solution-extension-create package outlines your options.

Once the project is created, it is time to create a solution extension (we'll also call them controls interchangeably for convenience in this paper). The following command creates a scaffold to do so:

```
npm run create:solution-control
```

You are prompted for a name and a control type. For the simplest and most common example, we'll choose "Page." This command creates a handful of files that we'll explore in the next major section. It also registers the new control—which is treated by Angular as a self-contained construct called a "module"—in a configuration file called app.module.ts. For now, the most important thing to know is that app.module.ts is your project's top-level module, and the one that gets translated into a minified pure JavaScript file that SAS Visual Investigator accepts as an import. Last, we'll run:

```
npm install @sassoftware/vi-api
```

That package contains TypeScript types and definitions of almost anything relating to SAS Visual Investigator and its Application Programming Interface (or API, which is a collection of functions we can call to cause SAS Visual Investigator to act or return information).

For more detail about the choice of "Page" two paragraphs ago, we'll return to the definition of a solution extension. They can go on standard SAS Visual Investigator entity pages, which is the build target we chose. As the

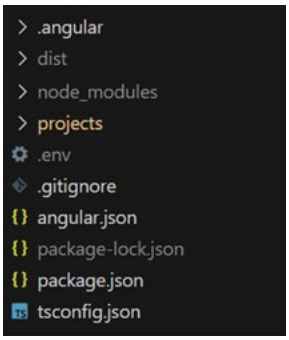
other options have hinted, they can also appear on homepages, on entity toolbars, and in property editors for both pages and toolbars. Just in case the terminology is unfamiliar, property editors are the panes that appear on the right side of the page builder interface when configuring a control, including those built into SAS Visual Investigator. They are also available in the "Edit Toolbar" window that appears when configuring toolbar items.

Structure

Folders and Files

Now, with the project set up and a control added to it, we can explore the structure and where our custom code goes. At the top level of your project, two files to note are .env and angular.json as shown in Display 1:

Display 1. Top level of a solution extension project



The .env file is the place to change information that you provided during project setup, such as hostname and login credentials. Angular.json is the configuration file for this instance of Angular. It's helpful to know because your upload budgets are configured there, and they are likely too small by default. It is a flattened file, so if you are interested in expanding it with a tool such as Prettier, you can navigate to:

```
projects.elements.architect.build.configurations.production.budgets
```

Alternatively, a quicker way to get there is to CTRL-F search the document for "budgets." Then switch the values that follow for maximumWarning (Angular warns you if your bundle is over this size) and maximumError (Angular does not upload bundles over this size) to fit your needs.

The bulk of the project resides further down in the structure. Relative to the root level, code files are in **`/projects/components/src/lib`**. A folder should be there with the same name as the control that you created earlier. Up one level from **`lib`** under **`src`**, there is also an app.module.ts file that we've already hinted is important. If you open it, your control should be registered in the "imports" array as shown here:

```
@NgModule({
  imports: [
    MyNewControlModule
  ],
  providers: [
    ***
  ],
})
```

If you want to add a control from another project to this one, you could do so by adding its module name to that list. You would need to copy its folder from under **`/lib`** to your project's instance of **`/lib`**, and then account for any dependencies. We'll discuss using third-party code later, but for now, you should know that any modules that you want to import for your project globally are imported here.

Upon dropping into the control's folder under **`/lib`**, five files are listed. They all start with the name of the control, so we'll refer to them by the portions that are unique. They'll each get their own subsection below.

component.html

Component.html is the structure file, or template. Right now, all it contains is an HTML paragraph tag for testing. To develop something more interesting, this is where the visual interface is defined. Anything you put here is wrapped in a custom tag (named in control.ts) to be placed on the SAS Visual Investigator page. This includes Angular specific bindings and tags such as ng-template, ng-if, and interpolation braces. Best of all, any variables that are declared as properties in component.ts can be referenced with event binding to update automatically (either through code or by prompting the user).

To demonstrate what we can do here, we can add a line below the existing <p> tag. The double curly braces invoke the Angular interpolation process, and it attempts to fill in the value of the matching variable declared in component.ts:

```
<p>Your admin wants to say: {{childNode.typeAttributes.userText}}</p>
```

The object referenced by childNode is a special one that we'll describe later in this section.

spec.ts

This file is where you can define automated tests to ensure that your code is running as expected. It hooks into the Angular suite of testing tools. We're not going to cover this file here, though, since testing in Angular is beyond the scope of getting a solution extension running.

component.ts

This file runs the action of your control. Any data that you want to transfer or manipulate goes through here, as does any interaction that you want to have with the SAS Visual Investigator backend. Display 2 shows how it looks right now:

Display 2. Component.ts file

```
import { Component, OnInit, Input } from "@angular/core";
import { Control, ControlMemberApi } from "@sassoftware/vi-api/control";
import { PageModel } from "@sassoftware/vi-api/page-model";

@Component({
  selector: 'sol-test',
  templateUrl: './test.component.html'
})
export class TestComponent implements OnInit {
   @Input() childNode!: Control;
   @Input() pageModel!: PageModel;
   @Input() controlApi!: ControlMemberApi;

   constructor() {
   }

   ngOnInit(): void {
   }

}
```

The first thing to note in Display 2 is the section with three input properties: childNode, pageModel, and controlApi. They are passed in from the page element to which this control is attached. Almost anything that you want to do with SAS Visual Investigator can be done through these three:

- childNode.typeAttributes is primarily useful to get the properties configured for this control in Page Builder.
- pageModel.data is how we update the values on the page, and pageModel.mode is how we check the mode.
- Last, controlApi is how we manipulate SAS Visual Investigator; for example, by forcing a save or a reload of the page.

They also all have their types defined with custom schematics provided by @sassoftware/vi-api, which we imported earlier. This means that a good IDE shows you the properties and methods of each as you type.

Finally, most code that you write goes in ngOnInit(). This enables your code to run once your control is ready to act. For further research into other options, though, these functions—and the interfaces they implement using the "implements" keyword—are called lifecycle hooks and are a common piece of Angular components.

control.ts

This file, shown in Display 3 below, defines elements of your control such as icons and names. Most importantly, it also contains an object at control.controlAttributes.attributes that defines the configuration options that are available to an admin user in Page Builder. We'll call these property editors. They are available using the name you give them as a property of the attributes object.

Display 3. Control.ts file

```
export const control = {
  category: "Fields",
  controlDescription: {
    defaultText: "test"
  },
  directiveName: "sol-test",
  displayName: {
    defaultText: "test"
  },
  name: "test",
  controlAttributes: {
    attributes: {},
    metadata: {
      renderAs: ControlType.WebComponent,
      states: {
        readOnly: true,
        required: true
      }
    }
```

An attribute entry defining a text field can be as simple as follows. For example, suppose a user wants admins to configure which text shows up on the page:

```
userText: {
  displayName: {
    defaultText: "Text to Display"
  },
  required: true,
  type: "textInput"
}
```

We'll examine the outcome of adding this entry to the attributes object in the deployment section. For more information about the structure of this object or which property editor types are supported, see the SAS Visual Investigator API Documentation or SAS Visual Investigator Help Center Documentation, respectively. The information in the second link is specific to SAS Visual Investigator 10.8 but still applies to SAS Viya 4.

In this file, the directiveName "sol-test" is included, which is what we can search for if we want to inspect our control using a web browser's developer tools.

module.ts

As described briefly in the Scaffolding subsection, Angular is designed so that almost everything is a module. This file defines your control's module and is imported by the app module. Although many controls wouldn't require editing this file, this is where you could import any helper modules that you write or provide any imported dependencies that are needed only by your control. For example, those tasks may apply if you design a pop-up form generated by your original control. They are done by listing the appropriate module or service in the imports or providers array, respectively. We'll discuss this more later, but dependency injection and services are two Angular paradigms that can help organize a larger web app—or component in our case.

Deployment

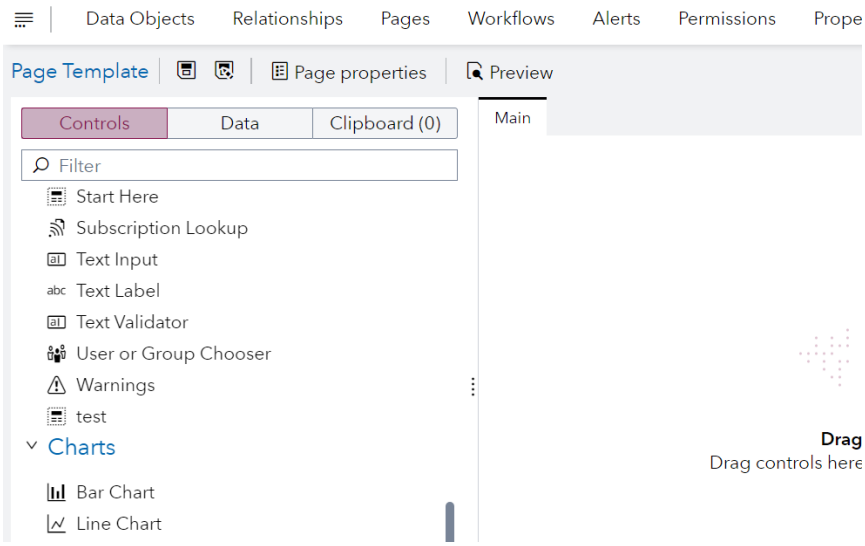
The Quick Way

With your code files ready, the last step is simply to run this code from your project root folder:

```
npm run watch
```

This code submits your control to SAS Visual Investigator and starts a process that watches your code for changes. If you save one of your code files with an update, it automatically resubmits until you end the script. Display 4 shows how things appear after the changes we made to the default files. You must clear your browser cache for changes to be visible:

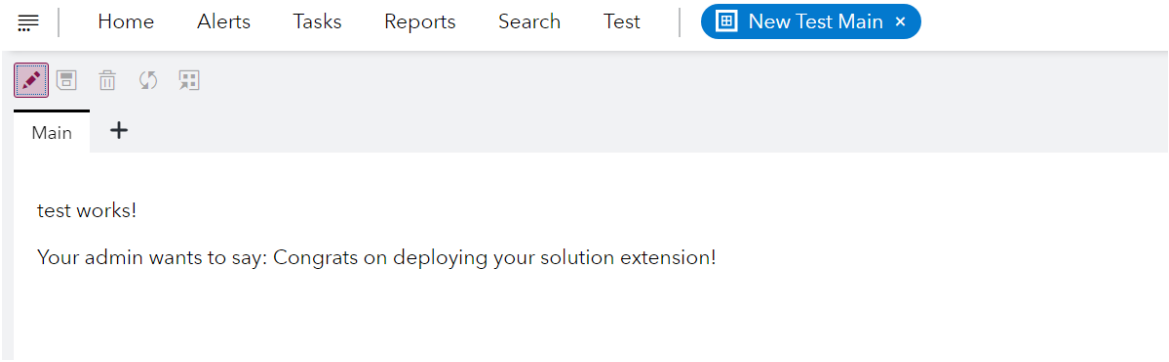
Display 4. Controls tab in page builder after changes



The control appears with the other SAS Visual Investigator controls on the left side of Page Builder. We can then drag it onto the page. Notice that our message doesn't quite work yet, since we haven't configured one to display. We can do that on the right side where we added a property editor. In general, it's good practice to design controls differently for design mode and create/edit/view modes. That way, users won't be confused by any missing information or not-yet-functional UI elements.

After saving the page, Display 5 shows our changes when viewing the entity:

Display 5: The control on an end user-facing page



The Classic Way

We won't go into much depth here because SAS recommends the previous option. However, to upload a control without using SAS scripts, you can access the fdhmetadata.dh_control table in the PostgreSQL database behind SAS Visual Investigator. The table is described with more detail in SAS Visual Investigator Help Center Documentation. Again, this link is written for 10.8 but applies in SAS Viya 4.

One key point for this paper is that a control made with Angular only stores code in the directive_txt field. To manually upload a code bundle, you need only to locate main.js in **`<project root>/dist/elements/`**, and then place the contents in directive_txt. Further, if using Angular, one row in this database represents an entire project. While a row otherwise generally represents just one control, consult the documentation for your framework to see how it bundles your code.

Importing or Removing a Control

To import an Angular control from another SAS Visual Investigator instance, as hinted at in the section on structure, you'll need the code that makes up its module. This is generally the set of files we discussed in that same section. Next, add that control to the imports array of app.module.ts, which using SAS scaffolding sits one level above those other files. Finally, there might be additional code dependencies for your imported control. A good IDE warns you that it can't find them, but you can also check the import statements in the control's module.ts and component.ts files. If third-party npm packages are missing, they can be fetched with

```
npm install [package name]
```

After those steps are complete, all you need to do is rerun the "watch" script if it is not already running.

Removal is even easier for an Angular control. All you need to do is remove the control's module from the app module imports array and let the "watch" script update. Of course, it's good practice to delete unnecessary files afterward to clean up, but that's not required. To remove an entire project from SAS Visual Investigator, you must delete its row in dh_control.

Other options for import of any control type include copying the dh_control row between SAS Visual Investigator instances or using the configuration import/export functionality in the SAS Visual Investigator admin interface.

In SAS Viya 4, projects are located under **`SAS Visual Investigator/Properties/Page Extensions`**.

To remove non-Angular controls, delete the row from dh_control.

Best Practices and Tips

This section offers some tips that come from both SAS R&D and our personal experience. Feel free to consider them in the context of your own solution extension project to hopefully save time and effort.

Organization

With controls now able to be bundled as packages, we recommend either grouping all controls for your SAS Visual Investigator instance together or separating them by their primary concern. Separation helps your system remain functioning well if a change to one project introduces problems, but it is at the cost of bundling sometimes large amounts of Angular overhead into each project. You need to weigh the best balance for your system.

The .env file in the project root stores passwords in plain text. Take care to properly protect this file if applicable.

Development

Use Angular tools available for your IDE to minimize mistakes and delegate the effort of knowing the name and location of every API function or variable that might be useful to you. We mentioned this tip earlier, but it's so helpful that it is good to repeat it.

SAS internal styles often make your controls look more professional just by adopting some of the same HTML classes. Use your browser's developer tools to see which ones are available.

If for any reason the controlApi in component.ts doesn't have something that you need, the full SAS Visual Investigator API is passed via the window. To let TypeScript know what to do with it, you can reference it with a type cast as

```
(window as sviWindow).sas.vi
```

Take advantage of the SAS Visual Investigator page modes to ensure that your control is robust in all four modes: create, design, edit, and view. Coupled with an NgSwitch directive, this can be relatively easy and powerful to do.

Deployment

Solution extensions can affect the performance of SAS Visual Investigator. It's good practice to test your control in a variety of situations.

The display name of your control isn't "pretty" by default. To fit the style of SAS Visual Investigator, you can rename it to have capitalization and spacing as appropriate in control.ts.

It's good for space and clarity to make sure that you are not keeping any unnecessary third-party packages, but

Angular only bundles the ones imported by your code. A good IDE indicates whether your code uses an import.

If you're collaborating with another developer and using a form of version control, the "watch" script is still an easy way to upload the latest version. SAS Visual Investigator overwrites old versions in Postgres based on names, so if the name matches, it should be successful. An especially intrepid developer might even find a way to pipe the code directly from a central repository. The SAS tools are open source so feel free to experiment!

References

Google. 2024. *Angular Docs.* Available at **angular.dev/overview**

SAS Institute, Inc. 2024. *SAS Developer Portal.* Cary, NC. Available at **developer.sas.com/sdk/vi/apiDocs/index.html**

SAS Institute, Inc. 2022. *SAS Help Center.* Cary, NC. Available at **documentation.sas.com/doc/en/vicdc/10.8/visgatorcc/titlepage.htm**

SAS Institute, Inc. 2024. *SAS Visual Investigator Solution Extensions: Getting Started.* Cary, NC. Available at **github.com/sassoftware/vi-solution-extensions/blob/main/docs/pages/1-getting-started.md**

Appendix 1: Angular for Solution Extensions

This appendix is designed to explain some of the Angular concepts and structures that are used in this paper. It's far from exhaustive, so we recommend consulting the official Angular Documentation to satisfy any further curiosity.

Components

Each control we build is an Angular component. These are essentially fancy HTML elements, but it might be easiest to think of them as the building blocks of an Angular app. As you saw with SAS Visual Investigator custom controls, they can have specific logic and styling and can even contain other components. And while it is beyond our scope, parent and child components can send data back and forth using @Input and @Output decorators similar to those we see in component.ts. Components that are more separated can communicate via services, which we discuss briefly.

The most important thing to note is that a component's template only has easy access to the data and variables that are stored in the component's class. Anything you'd want to display or collect as input from the user should have a home in the class, even if that "home" is just a function that fetches the information from somewhere else.

Directives

Directives are classes that make Angular work. They are the source of functionality such as DOM manipulation, and components are technically a type of directive. Angular outlines two types of directives besides components: attribute directives and structural directives. Building our own is beyond the scope of this paper, but there are a few built-in that can be very useful in building solution extensions. NgClass and NgStyle are attribute directives that can be used to reference variables containing data about which classes and styles to apply to an HTML element as described in Angular Documentation. They're applied with property binding, which is covered in the next subsection.

Structural directives of note include the following:

- NgIf – displays an element if the bound variable evaluates to a truthy value
- NgFor – displays a copy of an element for each item in a bound list

It's worth noting that these directives sometimes internally make use of <ng-template> to easily repeat snippets of HTML or <ng-container> for hosting directives that don't belong to any one element. The same Angular Documentation link in the previous paragraph also contains information for further learning about structural directives.

Interpolation and Binding

We already know a little bit about interpolation, which is invoked with double curly braces and attempts to evaluate the expression from the context of the component's class. This enables us to display dynamic content and—even better—updates automatically when the value changes.

Binding is similar, and it comes in two forms that can be active at the same time. The first is property binding, which ties properties of a DOM element (such as the src of an img element) to the value of a component variable. Property binding is a special case of interpolation. As a quick example, if we have a component variable called imageUrl, the following code creates a binding that automatically updates the img element if imageUrl changes. The bracket syntax invokes property binding:

```
<img alt="item" [src]="itemImageUrl">
```

Event binding is how Angular implements event handling. For example, to run an onClick() handler function when a user clicks a button, we can use the parentheses syntax and the name of the DOM event to register the binding. We can also pass $event to our handler function if we need more information about the event:

```
<button (click)="onClick($event)">Submit</button>
```

Event binding becomes even more powerful when combined with property binding in a construct that Angular calls two-way binding. Its use in custom components and directives is a bit beyond the scope of this paper, but we can make easy use of it in classic HTML form elements via the NgModel directive. For example, we can use combined syntax shown here:

```
<input [(ngModel)]="username">
```

This way, the username variable in our component always holds the most recent value that the user typed into the box. The reverse is also true, in that we can also instantly change the value in the box programmatically.

Dependency Injection and Services

Dependency injection is Angular's way of allowing code reuse. Usually, the reusable code is a "service," which for Angular is a class with a defined purpose. For example, a service can act as a central hub of information between two or more components, or it could be delegated tasks, such as fetching data. To make sure that the service is available, it must be both provided and injected. Writing our own services is beyond the scope of this paper, but to use a third-party service, import its module into your own module as appropriate. Then add the service to the relevant module providers array.

To inject a service into a component, just call for it in the class constructor. Usually this looks similar to this:

```
constructor(private myService: ServiceType) {}
```

Asynchronous Code

Angular strongly recommends an included framework called RxJS for its asynchronous code execution. Although the classic JavaScript promise paradigm works, Angular works better with a concept from RxJS: "observables." They can be used in similar ways to promises but have an extensive library of methods to combine them efficiently and can emit values as long as needed until complete. If you build a control that makes HTTP requests using the default Angular HTTP client, any responses come back as an observable. You might find, as we did, that you even enjoy using them more than promises. The primary documentation is an excellent resource for the curious, and it contains a helpful decision tree on which operator (function that combines observables) to use when. Last, simply passing a callback function to .subscribe() is usually enough to simulate .then() or async/await from plain JavaScript.

Release Information Content Version: 1.0 November 2024.