Paper AD09

Interfacing with Interfaces

Jeremy Teoh, Temeta Ltd, UK

ABSTRACT

Building user interfaces (UIs) can be a daunting task for a SAS® programmer. Communication between the front-end and back-end in real-time requires a stream via an API and/or a persistent data store such as a database. Setting these up often requires prohibitive effort, and any corners cut when developing a successful proof-of-concept will lead to extensive work down the line.

This paper shows how a set of generic function templates (an abstract interface) can be implemented by a library of adapters. The use of an 'adapter' layer between programs and data access methods serves to isolate code from the rest of the application, making programs developed this way much more portable. The function templates provide a framework for consistent access methods that can even share the familiar syntax across different programming languages.

The objective is to provide open-source data interface methods to streamline the prototyping of UIs. Development of the initial proof-of-concept UI is simplified using a predefined set of methods and a simple local flat file as its data stream/store. When the UI is ready to be scaled up (e.g. for enterprise use), the same front and back-end code can be used with any type of stream or data store by swapping out the data interface layer.

INTRODUCTION

Applications are great. Why aren't we seeing more of them developed in this industry?

Applications with well-designed user interfaces are the most enjoyable way of interacting with analytical software, and apps are getting increasingly easy to develop thanks to the plethora of tools and frameworks available. As programmers in pharma, however, we sometimes find ourselves in tightly controlled clinical programming environments and virtual desktops.

A programmer might have a great idea for how to streamline some laborious process by introducing a GUI, but lack access to the tools that would allow them to make a quick prototype. Without a prototype in place, however, the risk of wasted time (and pain of having to negotiate with IT) prove too much of a barrier to development.

From their manager's point of view, interface development takes too many resources. Ideas pitched by programmers to managers usually don't develop past the drawing board because of involvement of IT, admins and front-end specialists is too costly. Increasing the SAS licensing spend, spinning up servers and databases, and then venturing into the unknown with an unproven application is not an attractive prospect to those holding the purse strings.

How can we break out of this creativity-stifling catch-22? Any real solution would have to cover all types of application and work across a variety of environments. A solution in any one language is usually insufficient since different languages are good at different things, and an application's components do different things over the development life cycle. On top of this, we often have no say over the data ecosystem that we are working in. One answer to this problem is to focus on the *connections* between components rather than the components themselves.

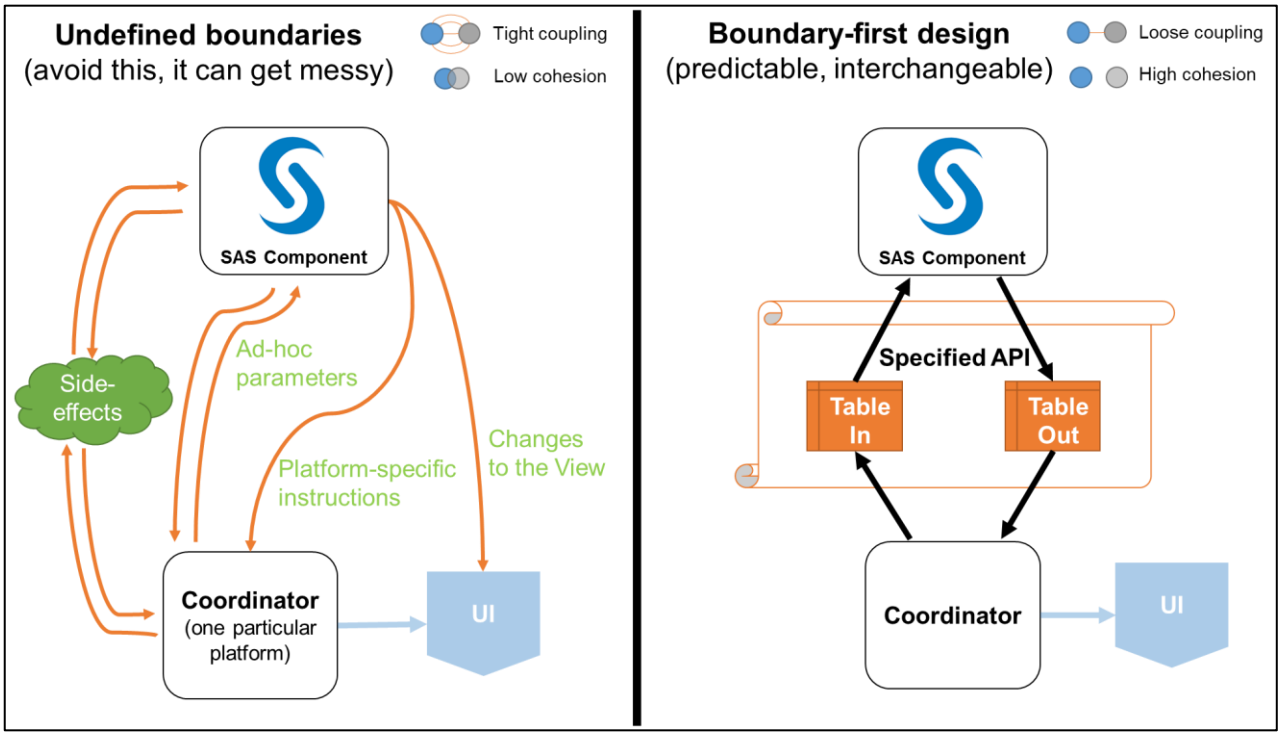
USING INTERFACES TO INTERFACE ACROSS INTERFACES

The word interface has several uses. Unpicking the various definitions here:

1. Interface (n) – the common boundary across which components of a system communicate with one another
2. Interface (v) - to connect in a compatible way
3. Interface (n, software) - an abstract definition of an object's behaviour e.g. inputs, outputs, and methods
4. Graphical User Interface (GUI) - the front-end display that the user looks at and interacts with
5. Application Programming Interface (API) – a specified set of rules that define how different components of a system communicate with each other. This could be different components in an application or even a way to expose an application's functionality to other applications

When developing applications such as GUIs (4), well-defined interfaces (3, 5) are a highly effective way to connect (2) components since their boundaries (1) are understood from the start. This helps designate the distinct responsibility of each component (cohesion) while preventing the components from forming unintended interdependencies (coupling).

An interface-based architecture is how to enforce decoupling by design. Boundaries between components get defined from the beginning. By agreeing on and documenting each component's inputs and outputs up-front, each component knows exactly which inputs and outputs to expect. Predefined interface methods instead of specific ad-hoc methods are used for API calls between components to ensure consistency. Planning for and incorporating these requirements from the start of a project helps to specify the components themselves while preventing accidental coupling during development – a step toward the ideal of high cohesion, low coupling.



Perhaps counterintuitively, decoupling is the best way to help parts of your application play nicely with one another. In a loosely-coupled architecture a component does not need to know the ins and outs of another component it is interacting with – instead only needing the *inputs* and the *outputs*. The less they know about each other, the better.

With this level of independence components can easily be emulated, tested and upgraded.

EASIER EMULATING

Since the inputs and outputs of components are known up-front, they can be used even if the components have not been built yet. You can emulate another component with a simple tool that produces similar output, or even by using dummy data in place of the other component's output. Up-front API specification and emulation of unfinished components allows parallel development.

EASIER TESTING

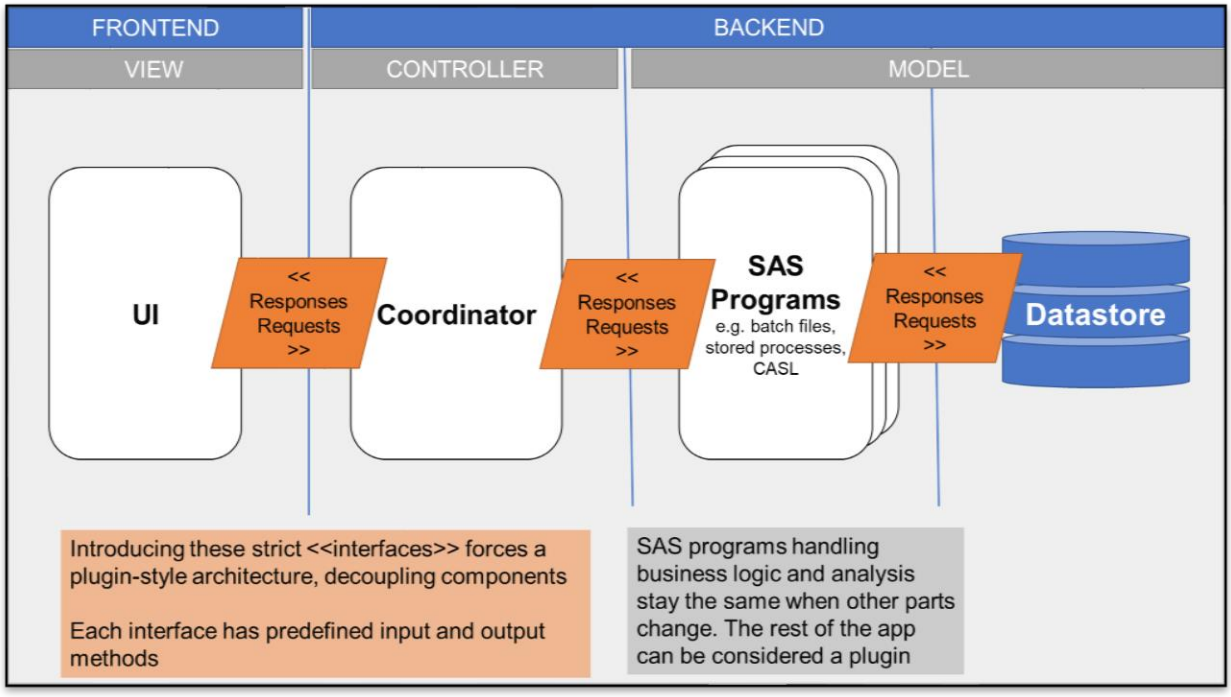
Testing becomes a lot simpler using interface-based architecture. Components can be validated using API testing and the test handlers can stay consistent thanks to the abstract interface methods. The most crucial advantages when it comes to testing is knowing what inputs and outputs to expect from a component and being able to test it fully in isolation.

EASIER UPGRADING

When upgrading from the prototype, particularly when having to scale up, new problems need to be tackled. This can involve having to purchase additional licenses, changing the datastore, and continuously improving the front end. When bringing the front-end out of prototype, the entire front-end platform might need to be migrated, potentially involving extensive modification to other components too. These are made more costly still by the programming time involved in integration and testing/debug cycles. By equipping yourself with well-specified interfaces, components can be swapped out and replaced as needs of the project change without the rest of the application noticing a thing. This is where the real savings in project time and budget are made.

COMPONENTS AND INTERFACES IN A SAS-POWERED GUI

Let's start with the general architecture used for graphical user interfaces, the Model-View-Controller (MVC) architecture. This is one of the oldest examples of application architecture, still widely used today because it works. With a clear separation between components in this way you avoid repeating code, resulting in high cohesion and low coupling.



Using a web app as an example: The *View* is the rendered HTML, the *Model* is the business logic and the *Controller* will act as a decision maker and intermediary between the model and view, keeping them separated by design while handling their instructions.

The GUI front-end should ideally be separated from the controller so that the controller's inputs and outputs are testable and easily replicated.
Protection of the application logic is also helpful - you don't want cosmetic changes to have any way of affecting the business logic. It can be helpful to consider the UI as a plugin to the business rules rather than the other way around.

When developing the GUI into a multi-user web app it can be prudent to ensure that SAS components can be developed independently from the front-end. The front-end developers do not need to worry about the business logic/analysis being performed by SAS, and the SAS developers do not need to know how to build the front end. Rather than requiring developers to acquire new skills, it is far more effective for developers with different specialisations to agree on a mechanism of communication and a well-documented API.

There are other advantages to keeping SAS processes strictly separated from the rest of the application

• SAS licenses and proprietary programs are not visible to the end user and only the appropriate results need to be rendered. Only a strictly predefined set of information can be passed between isolated sections of the application.

• SAS teams can work and develop independently from the front-end team, with clear demarcation of their responsibilities and an understanding of what to expect from the other side

• Work can progress in parallel since up-front documentation of the interface allows emulation of input from other sections of the application, even if they have not been completed yet.

ADAPTERS CAN PROVIDE A FRAMEWORK FOR APPLICATIONS THAT COMMUNICATE WITH SAS

To help more prototype applications see the light of day I wanted to find and/or build a generic interface that could apply to as wide a range of applications as possible. Apps come in all shapes and sizes, but what all SAS-powered apps have in common is a means for SAS to talk to the front end. If a working interface can be provided from the start, the rest of the app can be built around it as a set of interchangeable components. The business logic, front end, and documentation might vary from app to app but what can be standardised is the mechanism used to allow communication between SAS and other parts of the application

Viya® is one of the latest examples from several solutions from SAS themselves, and it comes with a connectivity suite including APIs for control from other programming languages. Judging by the capabilities of its Cloud Analytics Service and interoperability options, it provides an attractive option for future application development. The catch (and it is a significant catch) is that you need to have SAS Viya licenses in order to do this.

If you don't have Viya available to you, you might have access to a SAS Stored Process Server. There are many examples of web apps that have been built with Stored Processes and expert guidance[5] is available.

An example of interface-based thinking is Boemska's HTML5 Adapter for SAS (H54S)[1][3]. The team at Boemska have created an adapter that allows HTML5/JavaScript web apps to incorporate SAS stored processes. The adapter takes the form of SAS macros and JavaScript libraries that handle the connection between SAS and JavaScript by essentially allowing SAS datasets to be fed from JavaScript to SAS, and vice versa, in real-time.

Their solution is brilliant for several reasons, but I'll focus on the following here

1. The inputs and outputs are flat tables in JSON that translate directly to and from SAS datasets. SAS processes therefore effectively only deal with input datasets and output datasets, which is a form native and familiar to all SAS programmers.

2. It allows the latest and greatest web technologies (i.e. HTML5 and front-end frameworks such as Angular and React) to interact with SAS. The application front-end can be developed independently if needed from SAS processes by programmers with different specialisations. This makes a lot of sense for enterprise application development where it may prove tricky to find a front-end web app developer with clinical SAS experience, or find a SAS programmer with web app development experience. The functionality of H54S lends itself to a framework for SAS web application development - Boemska have developed an 'AppFactory' powered by H54S that makes it easier still to build web apps that make use of SAS Stored Processes.

3. Both SAS and JavaScript sides of the adapter are open-source and have been made available under GPLv3.0*. Not only does this encourage others to make use of their framework, the quality of the code benefits from enhancement and testing. Those making use of it can give back by making and sharing improvements. Use of

and collaboration on open-source standards and adapters such as this could go a long way toward eliminating the silos that we keep stumbling into in this industry.

* a commercial license is also offered for those who don't like GPL. More on open source licenses later in the paper.

A COLLECTION OF ADAPTERS COULD IMPLEMENT A GENERIC INTERFACE TO CONNECT SAS TO MULTIPLE OTHER TECHNOLOGIES

H54S solves the problem of connecting SAS to web apps but it has specific requirements for HTML5, JavaScript, and systems that have access to SAS Stored Processes. How can this solution be used to help pharma SAS programmers who do not have the necessary skills and licenses available to them?

Overcoming the initial disheartenment of finding out that someone else had done a much better job than I could have, I began to get inspired by the elegant design of H54S. Taking a leaf out of the Object-Oriented Programming definition of 'interface', consider what happens if you treat H54S as an implementation of a more general abstract interface between SAS and some other external technologies.

From the point of view of the non-SAS components of the application, SAS processes are called on demand and can be supplied any number of datasets via your chosen protocol.

SAS runs and returns one or more datasets to the external component, which then uses the content to determine the next actions the application should take.

SIMPLE SAS PROGRAMS THAT WORK ACROSS DIFFERENT APPLICATION PLATFORMS

From the point of view of the SAS processes, all programs will follow an identical, simple framework. One or more datasets can be specified along with the process when it runs. A standard macro provides access to the inputs as SAS datasets, so it is trivial to access them and use their content for conditional operation. This can all be in SAS BASE if needed. No additional expertise is required for these SAS processes to be written even though they are part of an application leveraging multiple other technologies.

The results of the SAS process are written to one or more SAS datasets, then fed back using a standard macro

```
All SAS processes called by the application follow this structure:
1. Get input datasets from the Controller
2. Do SAS things based on input dataset, create output datasets
3. Send output datasets back to Controller
```

The goal is a framework for SAS app development consisting of generic functions with an accompanying collection of open-source adapters allowing them to plug in to different technologies. Methods that allow SAS to interface nicely with other languages or platforms can be collated into a library of adapters that implement generic methods.

Each connection type consists of a SAS macro implementation paired with a library of methods to implement the generic interface from the other side. Prototype apps using this framework and pre-existing adapters can be developed rapidly, constructed using components that already have their connections documented and are easily migratable. The application could even be handed to app developers who don't need to know how the SAS side works.

All programs following this framework can use a consistent set of functions even when using a completely different connection. A generic SAS interface is implemented by different adapters, so only the adapters need to be swapped rather than changing code.

GENERIC INTERFACE FUNCTIONS FOR SAS

```
%macro getStreamDset(obj=, outdset=);
```

Deserialise the application's input to the SAS procedure into a SAS dataset. *obj* is the name of the object (e.g. if the input is in the form of multiple JSON tables, the name of the table). *outdset* is the dataset created from the content.

```
%macro setStreamDset(obj=, indset=);
```

Take the dataset *indset* that was generated in the SAS process, serialise it to object named *obj*, and feed it back to the application.

```
%macro setupDatastore(name=, targ=);
```

Establish a datastore connection. *name* is the reference (up to 8 letters) given to the created datastore and *targ* is a string (e.g. a name or query) describing which target in the datastore is selected.

```
%macro teardownDatastore(name=);
```

This cleans up and closes the named datastore connection as necessary.

```
%macro getDatastoreDset(name=, obj=, outdset=);
```

Read table *obj* from the named datastore name and put its contents into a dataset *outdset*

```
%macro setDatastoreDset(name=, obj=, indset=);
```

Take dataset *indset* and write it to table *obj* in the named datastore name

HOW TO FACILITATE PROTOTYPES USING THE SAME FRAMEWORK

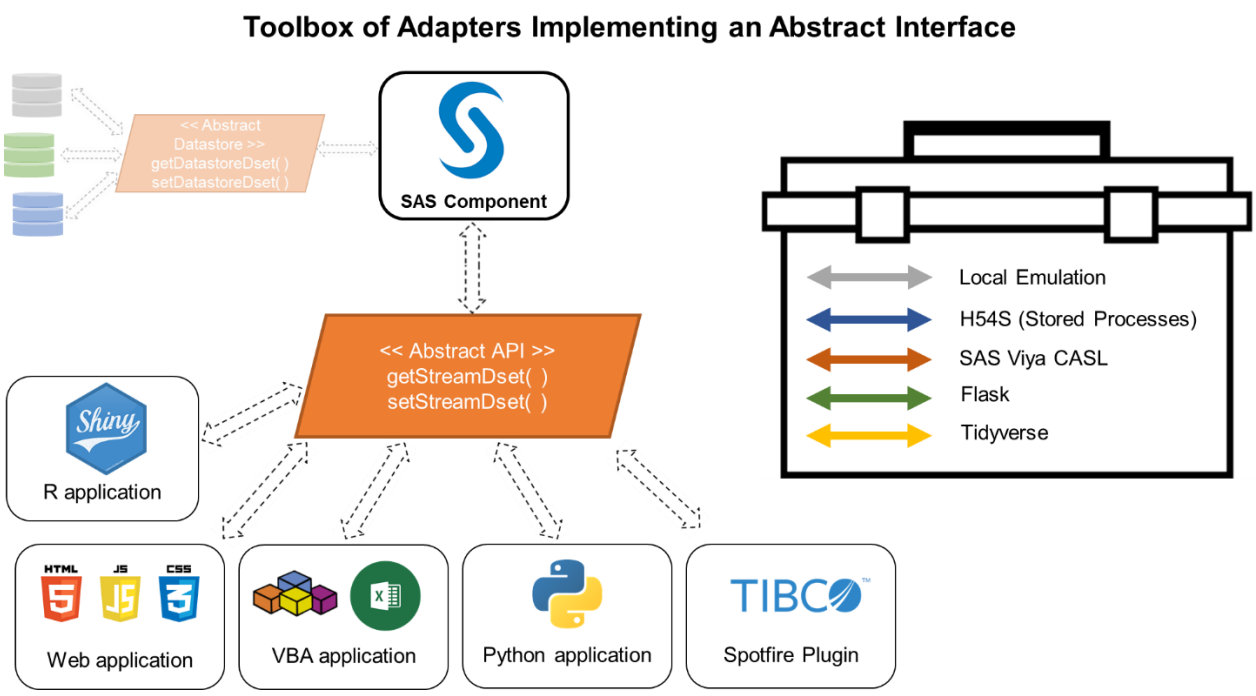
H54S is the prime example and works great for application developers. But what if you don't have access to a Stored Process Server, don't know your way around JavaScript or don't have administrator access to your SAS environment?

Finding a way to emulate this functionality could provide a way for pharma SAS programmers to begin developing prototype applications in a short time.

Local adapters that implement the common interface would support a working prototype, with a front-end developed using whatever available tools the programmer is most comfortable with (an open source collection of adapters handles the connections across different technologies and protocols). This way the same prototype could be upgraded at will using components following the same framework. By using generic functions in your code combined with adapters that implement them, the cost and labour of scaling your app up to different technologies becomes trivial rather than prohibitive. The only code changes required are to change which pre-existing adapter library is being pointed to.

When upgrading doesn't require any rewriting of code, use of a temporary stopgap technology in your prototype can become an advantage (since it was quick to develop) rather than a disadvantage.

Your prototype can be handed over to specialist application developers who can use the SAS components of your prototype without needing to modify or even have visibility of them. Since you are using a predefined general framework, all they would need is accurate API documentation and if possible, some dummy data.



With the adapters that have been made at the time of writing (find an up-to-date list on the sas-appfitter project's Github[2]) a programmer with no knowledge of JavaScript and access to only SAS and Excel® could knock together a VBA Userform that can talk to SAS locally via an adapter module. If necessary, this prototype can be handed over to specialist application developers who can take it further, for example turning much of the same code into a web app using H54S. The SAS programs themselves can be left untouched since only the adapter layer would need to be swapped round.

Armed with a prototype to demonstrate and a clear roadmap for scaling it up, a pharma programmer will be in a stronger position to control and justify any extra expenditure required to fully develop their application.

The figure above shows a few of the many ways that you could build an application that leverages SAS. As new solutions that make SAS applications more interoperable are created, a way is needed to bring them all together. In an ideal world, a company's SAS programs should work everywhere rather than being tied to any one solution stack; an abstract interface might be just the ticket.

OPEN SOURCE IS NOT SCARY

Understand and embrace open source solutions! Save yourself the plumbing between components by using tried-and-tested frameworks. If improvements or new adapters need to be developed these can be contributed back to improve the solution further. There is no point constantly reinventing the wheel when you could be sharing in the best solutions out there.

Most open source licenses allow you to use the code commercially for free, provided that the correct credits and licensing are applied to the distributed end-product. In the spirit of open source, it is fair to contribute back the modifications and improvements made to this code.

Some licenses such as the GNU General Public License version 3 are copyleft, meaning that anyone who has your software distributed to them has the right to see the source code of all of the software that they receive, even if only a tiny part of it is licensed under GPL. In the pharma world we are used to working in highly regulated environments

with everything under lock and key. The risk of viral licensing being injected into a system is not one to be taken lightly.

If any of your application's functionality needs to be used outside of your company and your company does not want to distribute the source code, there are legal considerations. Managers in this industry are sometimes very frightened of these, and not without reason, but the fear mostly comes from a lack of understanding of the license.

The most popular open source licenses allow (monetarily) free use, even commercially. The restrictions only come into effect when you distribute the application while hiding the source code or claiming that it is yours.

The GPL v3 makes this explicit: *To "convey" a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.*

You can use interfaces to expose your app to the outside via API without being required to make the back-end source code available. Application Service Provider commonly used in the Software as a Service model (SaaS). You must make sure that any code (such as JavaScript) that gets loaded on the client side is not licensed under GPL or similar. A minority of licenses such as the *Affero GPL* (aGPL) go as far as to insist that even server-side code must be visible, but these are not as popular due to the impracticality of such a requirement.

Please note that I am not a lawyer, so this paper does not constitute legal advice. Do your own research and be fully aware of the licensing implications of code that you use.

CONCLUSIONS

An interface-based architecture is an effective way to create robust, testable, well-documented applications that work with a variety of technologies. Being restricted to SAS for analysis and data access does not mean that other languages should be avoided. Adapters that interface SAS with other platforms facilitate applications that have the best of both worlds.

Proprietary solutions exist but not everyone has access to them. By embracing open source, we can develop a collection of interoperability adapters and seed apps as a community. A common framework leads to increased consistency and compatibility between components built for different applications by different people.

H54S has been generalised into an implementation of a framework for developing applications powered by simple SAS programs that input and output SAS datasets. The next adapters developed to extend this framework emulate the same functionality locally so that dependency-free prototypes can easily be developed. These local adapters implement the common abstract interface so the same prototype can be migrated to enterprise-grade implementations such as H54S without modification to the SAS programs. Further connection types are being incorporated into the same framework to support other application environments.

Lowering the barriers to application prototype creation and subsequent development may empower pharmaceutical SAS programmers (particularly those who grew up with the internet and an appreciation for UX, but lack experience in pharma platforms) to transform their ideas into applications for their clients and colleagues to use.

REFERENCES AND RECOMMENDED READING

[1] HTML 5 Adapter for SAS (H54S) - https://github.com/Boemska/h54s

[2] Abstract interface adapter collection (sas-appfitter) - https://github.com/TeMeta/sas-appfitter

[3] SAS macro collection (Macrocore) - https://github.com/Boemska/macrocore

[4] Nikola Markovic “Turn your SAS® programs into Apps using HTML5 and H54S” PhUSE 2017 AD02 - https://www.lexjansen.com/phuse/2017/ad/AD02.pdf

[5] Phil Mason “Developing Web Applications with SAS® Stored Processes” SAS Global Forum 2014 Paper 1272-2014 - http://support.sas.com/resources/papers/proceedings14/1272-2014.pdf

ACKNOWLEDGEMENTS

I would like to thank Nikola Markovic, Allan Bowe, and Giuseppe Di Monaco. Their advice, support and technical insight were invaluable while creating this paper.

CONTACT INFORMATION

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

Jeremy Teoh
https://www.linkedin.com/in/jeremyteoh/
Tel: +44 776 995 0406
Email: jeremyyjteoh@gmail.com