GenApp, Containers and Abaco

Technical Paper

Emre Brookes*
The University of Texas Health Science Center
San Antonio, TX
brookes@uthscsa.edu

Joe Stubbs
Texas Advanced Computing Center
Austin, TX
jstubbs@tacc.utexas.edu

ABSTRACT

GenApp is an NSF-funded framework for rapid generation of applications including feature rich science gateways. GenApp is being successfully used to produce science gateways wrapping scientific programs. Its organization is designed to simplify the process of adding new features and capabilities to generated applications. A limited set of definition files define application generation. To bring a new executable into GenApp, one creates a single "module" definition file. The executable must run on some compute resource accessible by the generated application. Installations of the executable on target resources may be complex. To simplify portability of execution, we introduce automatic containerization of defined modules and integration of container execution. Abaco is an NSF-funded web service and distributed computing platform providing functions-as-a-service (FaaS) to the research computing community. Abaco implements functions using the Actor Model of concurrent computation. We introduce GenApp integration of execution with Abaco as a resource.

CCS CONCEPTS

• **Software and its engineering → Software design techniques**; **Software development methods**;

KEYWORDS

Science gateway, Container, Actor

ACM Reference Format:
Emre Brookes and Joe Stubbs. 2019. GenApp, Containers and Abaco: Technical Paper. In *Practice and Experience in Advanced Research Computing (PEARC '19), July 28-August 1, 2019, Chicago, IL, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3332186.3332191

1 INTRODUCTION

In this paper, we begin with background on Containers, Abaco and GenApp. Subsequently, we provide details on new developments. GenApp has been installed into a container and extended to build containers from defined modules. Additionally, GenApp has been extended to support execution on Abaco resources. These developments enhance the portability of execution and installation by using container technology and Abaco resources. Once the structure of GenApp has been understood, it is a straightforward process to add new capabilities as we will demonstrate in this paper which could be used as a template for future novel compute resource targets.

2 BACKGROUND

2.1 Containers

Linux container runtimes leverage features of the operating system kernel such as cgroups and namespaces to provide process-level isolation for applications. While virtual machines virtualize hardware interfaces and contain a complete operating system running over a hypervisor, containers are userland processes sharing the underlying host's kernel. As a result, the start up time for a container tends to be much shorter, on the order of 100-200 ms, compared to that of a virtual machine, which can be on the order of minutes. It is common for containers to run within a rooted file system and isolated network stack, making them self-contained, executable packages whose only dependency is the container runtime. As a result, containers can be used to increase application portability with a minimal performance tradeoff.

2.2 Abaco

Abaco (Actor Based Containers, [1]) is a distributed computing platform funded by the National Science Foundation (OAC-1740288) based on the Actor Model of concurrent computation and Docker[1] containers. The Actor Model is a theoretical model of computation in which "actors", the computational primitives, receive messages addressed to them. In response to receiving a message, an actor can: 1) perform a computation and save state, 2) send messages to other actors, and 3) create new actors. An individual actor can only affect its private state: interaction with other actors is limited to sending them messages. Since the number of actors can grow throughout the execution, and since individual actors are independent, the Actor Model is inherently concurrent.

In Abaco, users associate actors with Docker container images, and the Abaco system generates a unique HTTP URI for each such registered actor. Once an actor has been registered, an agent can send a message to the actor by making an HTTP request to the actor's URI. For each message sent to an actor, Abaco executes a container from the actor's image, injecting the message data into the container, either as an environment variable in the case of text data, or over a Unix Domain Socket, in the case of binary data. Abaco will queue additional messages received for a given

*Corresponding author.

PEARC '19, July 28-August 1, 2019, Chicago, IL, USA

ACM ISBN 978-1-4503-7227-5/19/07…$15.00
https://doi.org/10.1145/3332186.3332191

[1] https://www.docker.com

actor and execute containers accordingly, as resources become available. Actors can leverage a state property to store arbitrary data across container executions; alternatively, actors can be registered as "stateless" enabling Abaco to start multiple actor containers in parallel. See Figure 1.

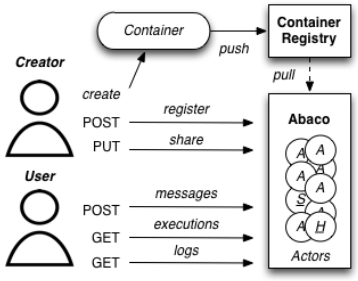

Figure 1: Abaco usage workflow: Creators build functions deployed in containers, send them to a public registry, then create and share actors via web service calls. Users execute functions by making web service calls that send a message to a specific actor. Figure © 2019 Joe Stubbs

Abaco actors are empowered to register new actors and send messages to existing actors, just as in the traditional Actor Model. More generally, the actor can make authenticated API requests to any TACC API by utilizing short-term OAuth access tokens injected into the actor's container at startup by the Abaco system. Actors can also be configured with POSIX interfaces to high performance storage within the TACC datacenter, such as the global parallel file system, Stockyard, and Corral, the data collections storage system. In these cases, actor containers are launched using the UID and GID associated with the owner of the actor so that file access is in accordance with the permissions on the underlying file systems.

2.3 GenApp

GenApp [2–4] is a tool developed under the international CCP-SAS [5] project, jointly funded by the EPSRC (EP/K039121/1) and NSF (CHE-1265817). The grant focused on advances in small angle scattering software and included a requirement to expose a diverse collection of software via a web portal. The initial software suites consisted of programs written in Python, C++ and Fortran. We needed a way to wrap them into a full featured web interface supporting multiple execution methods and requiring minimal short and long term effort to develop and maintain. As we did not relish maintaining a new collection of hand written code and finding no satisfactory extant tools for this purpose, we created GenApp. The primary goal of GenApp was to simplify creation of applications wrapping collections of existing program modules. GenApp is target language agnostic, and is designed from inception be able to build applications on a variety of targets. Currently our "html5" web based target is our most capable target language, but Qt variants and Java GUI targets are also available and planned for advancement. GenApp has successfully been used to develop multiple science gateways[2] and has since received dedicated NSF funding for further development.

GenApp generated science gateways support many features, including multiple job execution models, an integrated server based file system (to allow reuse of input or output files), OAuth support for login, messaging, context sensitive help (module and field based help), full job history with the capability to attach to running or previously run jobs, administrator mode with user management, project management, integrated context sensitive feedback[3], integrated plotting (2D, 3D and a vast variety of others[4], and interactive atomic structure display[5]. Additional features are added as needed by use cases. Websites generated are typically hosted on a VM, be it on a developer's laptop, dedicated host, or cloud resources such as NSF/Jetstream [6], TACC/Rodeo[6], or AWS where prepared images are available[7].

The extensible variety of current execution models include running on local or managed compute resources such as those available from NSF/XSEDE [7]. GenApp integration with Apache Airavata for the application execution has been prototyped [3]. Apache Airavata [8] is a software framework that enables one to compose, manage, execute, and monitor large scale applications and workflows on distributed and queue-managed computing resources such as local clusters, supercomputers, computational grids and clouds. OpenStack [9] as a target resource with optional job-specific XSEDE project accounting has been integrated into GenApp [10], to support efficient elastic cloud computing on NSF Jetstream.

GenApp documentation is available on our web-site[8]. Some of the material covered later in this section, particularly, deploying and modifying the "Energy Calculator" application (Fig. 3) is available online[9]. For more information on additional training or for any questions, interested individuals can subscribe to the users' mailing list[10].

2.3.1 Organization. The generation of applications is driven by four primary types of definition files as shown in Fig. 2. Definition files simplify utilization by being the definitive reference for all configuration options. The one directives file is the entry point for generation and contains overall application information including a list of target languages to generate. Each target language, such as "html5" for science gateways, has its own definition file which contains assembly instruction details. The menu definition file describes the user facing organization of underlying component modules, and thusly references needed module files. Each module definition file describes an underlying executable program. The module file can be thought of as an interface description language (IDL) describing inputs and outputs combined with a user interface description language (UIDL) describing an interactive user interface

[2]SASSIE-web https://genapp.rocks/sassie2 has over 600 registered users and has run over 20k jobs in 2018. Other GenApp generated gateways and learning materials can be reached from https://genapp.rocks
[3]The job input and output objects can be automatically attached to simplify user support.
[4]Limited support for Flot, greater support for Bokeh and full support of all Plotly types.
[5]JSMol and NGL
[6]https://www.tacc.utexas.edu/systems/rodeo
[7]https://genapp.rocks/get
[8]https://genapp.rocks
[9]https://genapp.rocks/learn (see "GenApp Basics" tutorial)
[10]http://genapp.rocks/join

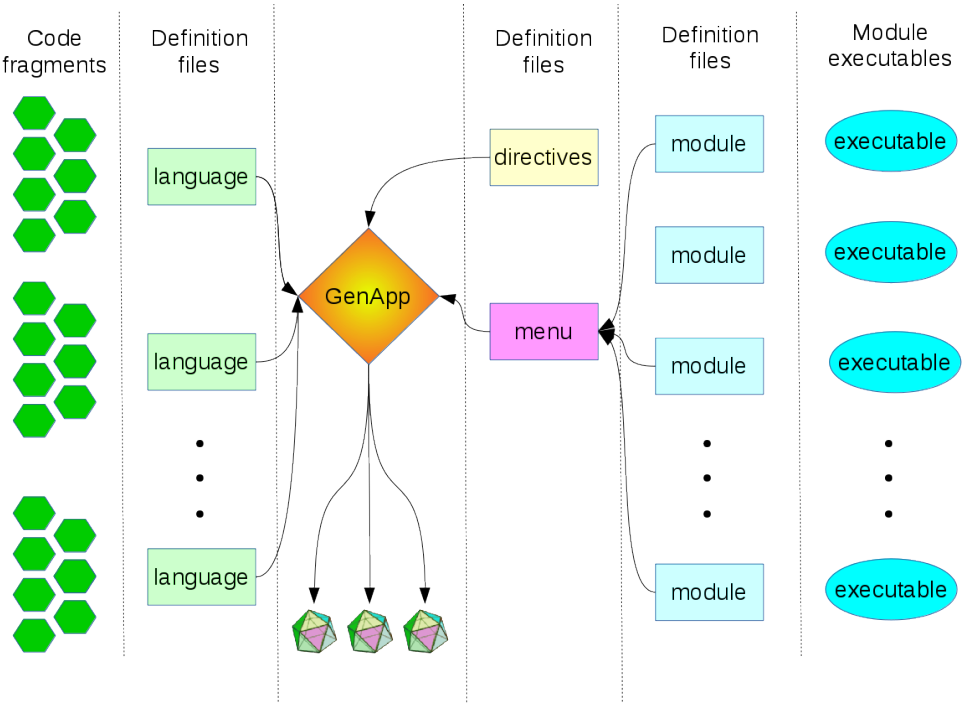

Figure 2: GenApp organizational overview. GenApp processes the definition files to generate applications. The language definition files contain target language specific assembly instructions including references to code fragments which are modified assembled to produce the applications (icosahedra). Figure © 2019 Emre Brookes

along with any additional information needed to install or run the executable.

The running of generated science gateways is supported by two additional definition files which can change dynamically[11]. The application configuration file contains information such as IP addresses and available execution resources. The secrets file contains any required long lived passwords or tokens.

In summary, the definition files are: directives; menu; module – per module/executable; target-language – per target language; appconfig; and secrets. All definition files in GenApp are JSON [11] formatted. Due to its popularity with web development, JSON parsers are available for most languages. JSON contains nested keys and values.

To simplify exposition, we refer to values using a definition-file:key notation, e.g. a module's executable value will be referred to as module:executable. Additional ":key"s may be appended to the notation for nested values. When context is clear, values may be referred to simply with the final :key.

2.3.2 Definition files. To create an application, a researcher must once create the directives global definition file describing the application attributes such as title, default colors and the set of target languages to process. The researcher's primary hurdle to bring a new executable into GenApp is to properly create the module definition file and wrap or modify their executable to accept input and produce output as defined in their module definition file. Additionally, collections of modules are organized in the menu definition file. Once all definition files are properly created, applications are generated by running the GenApp command line program which produces working instances for all directives:languages.

A module is some defined executable within GenApp. All input and output to a module's executable (module:executable) are JSON objects. An executable wrapper can be written in any language that can convert the native input and output to JSON. The module definition file (Fig. 3, left) contains all information about the module. This, of course, includes all input and output fields, module:fields. Each field is uniquely defined with an ID. In addition, the attribute key for each field is the module:field:type, which can take values such as "integer", "text", "plot", "atomicstructure", etc.

For a simple example, suppose one had the following Python code:

```
def einstein(mass, speed_of_light):
energy = mass*(speed_of_light ** 2.0)
return energy
```

To bring this code into GenApp, one needs to write a JSON text file describing the input and output as shown in Fig. 3. Next, the module developer must either modify the executable's code or write a wrapper script to accept JSON input from the command line and appropriately map the module defined input variables to local variables or known inputs of the executable. Similarly, the executable's output must be mapped to JSON as specified in

[11] after GenApp generation of the application.

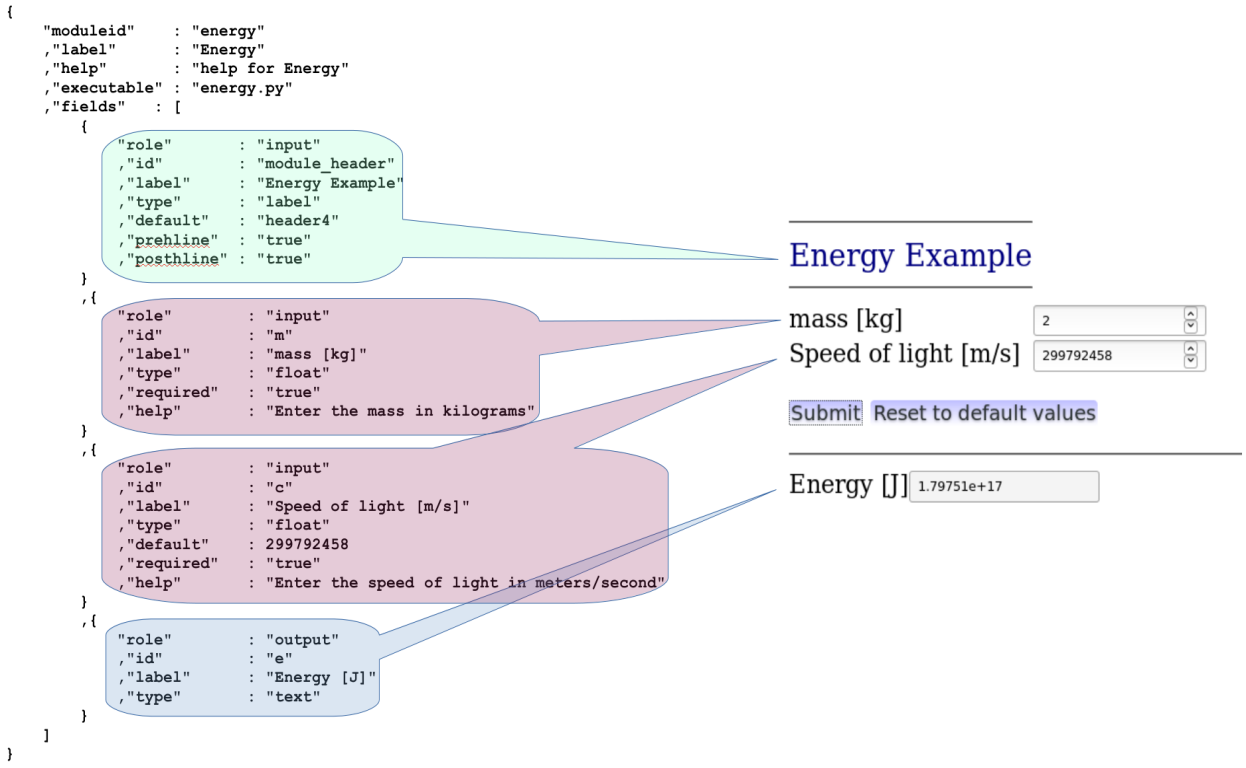

Figure 3: An example of a GenApp module definition file wrapping a Python executable (left), and the generated application UI (right). The module file contains JSON text describing input and output. Each module:fields value corresponds to a UI element as indicated in the call-outs. Figure © 2019 Emre Brookes

the module definitions. More details on wrapping executables in GenApp are provided in [4].

2.3.3 Execution model. During the running of an application, a user navigates to the specific module, fills out the input fields in the UI and submits the job. The application assembles the user's input into a JSON input object which is passed as input to the executable. The executable processes this input and produces a JSON output object which is returned in its output. This output is eventually passed to UI which populates the output fields in the UI. The specifics of execution handling are target language dependent. In this case, we consider the directives:language "html5" for generated science gateways. Available and default compute resources are defined in appconfig:resources. An example extract of resource information from appconfig follows:

```
,"resources" : {
  "local" : ""
  ,"compute0" : "ssh compute-0-0"
  ,"compute1" : "ssh compute-0-1"
  ,"airavata" : {
    "run" : "airavatarun"
    ,"properties" : {...}
  }
  ,"oscluster" : {
    "run" : "oscluster"
    ,"properties" : {...}
  }
}
,"resourcedefault" : "local"
```

In this extract, appconfig:resources:local runs on the web server and can be useful for low overhead execution. The entries appconfig: resources:compute0 and :compute1 give examples of simple ssh accessible resources. Any valid shell command can be specified and will be prefixed to the command line starting the executable[12]. If the value of the resources entry is an object, the :run value of the object will be used as the command line prefix, e.g. appconfig:resources:oscluster:run. This case is required for resources such as "airavata"[13] and "oscluster"[14] that require additional properties defined.

To add a new resource, one simply needs to add a new entry to appconfig:resources. A module definition file can specify a preferred resource in module:resource, otherwise, the global appconfig:resourcedefault will be used.

2.4 Related Work

The Abaco platform can be compared with two broad classes of offerings: containers-as-a-service and functions-as-a-service. Amazon's

[12]For example, a load balancer or a program to allow the user to choose a resource could be used.
[13]Queue-manged resources
[14]Elastic computing via OpenStack

Elastic Container Service, Google's Container Engine and open source alternatives like Apache Mesos and Kubernetes provide container orchestration on cloud infrastructure. These systems excel at scheduling and scaling stateless microservices packaged into container images. More recently, commercial functions-as-a-service offerings such as Amazon Lambda and Google Cloud Functions as well as open source projects like OpenWhisk and OpenFaaS have emerged to provide on-demand execution of singular functions.

Abaco provides functions-as-a-service but with Actor Model semantics, where actor executions can return results and state can be persisted across executions. By allowing arbitrary Docker images for functions, Abaco provides a more flexible execution environment than that offered by services such as Lambda or Google Cloud Functions, where the execution environment is limited to a small number of predefined language runtimes. While most other functions-as-a-service platforms expect messages to be executed as they are received and in parallel, Abaco's internal queuing system can buffer tens of thousands of messages to a single actor and can thus support long (on the order of hours) executions for stateful actors. Additional Abaco features such as POSIX interfaces to high-performance storage and access to more memory and CPUs for actor executions make Abaco a better fit for research computing workloads.

We know of no other framework apart from GenApp which can build applications to an extensible variety of target languages. The closest match of which we are aware is HUBzero's Rappture[12] toolkit which reads XML files to build tools for HUBZero, however, our experience was that HUBZero did not provide a navigable and controllable "application" environment as might be expected in a GUI environment, rather a collection of separate "tools".

3 NEW DEVELOPMENTS

3.1 Containerized GenApp

GenApp itself can now be deployed in a container. A container's natural isolation allows transportability of GenApp. We recently utilized this for integration with the NSF-funded Seedme2 project[15] where GenApp is co-hosted with Seedme2 in separate containers. An Ubuntu based container image is hosted on DockerHub and can be accessed via `docker pull ehb1/genapp`. Usage instructions are available on https://genapp.rocks/get. For hosts running Docker, this is the easiest way to test or run GenApp.

To build a Docker container image, Docker reads a "Dockerfile" which contains a sequence of commands for building the image, including the base image, and installation of any packages or special commands. GenApp contains a script for installation, so the work to implement this was translating these install steps into a Dockerfile.

3.2 Per-module containers

GenApp now supports generation of container images from defined modules with the new target language "docker". To achieve GenApp generation of module specific Docker container images, the Dockerfile must be built from the module definition file, which required extending the GenApp module definition file to support dependencies. Once such images are created, Docker can be used as a compute resource in GenApp's "html5" target language.

To add dependencies to a GenApp module definition file, the module:dependencies key is added. module:dependencies is an ordered list of JSON key value pairs. Available keys for module:dependencies are:"base", which can be any available image name; "file" which will add files; "env" which will set environment variables; "run" which will run a shell command; and also various helper tags to simplify installation of packages such as Python-pip, Python-conda and Perl-CPAN. The full set of supported dependency commands are available. [16].

For a Perl executable example, the following could be added to the module file:

```
,"dependencies" : [
  { "base" : "perl" }
  ,{ "cpan" : "JSON" }
]
```

For a Python executable with two file dependencies, one could add:

```
,"dependencies" : [
  { "base" : "python" }
  ,{ "file" : [ "filename1", "filename2" ] }
]
```

Note that the module:executable is automatically included and does not need to be specified as a module:dependencies:file. The task of extending a module definition to include dependencies is relatively straightforward, and any module developer should be aware of the dependencies of their executable. Nevertheless, it should be possible to provide a utility to interrogate an executable for dependencies, which would help simplify setup of module dependency information.

To build the Docker container images, one simply adds "docker" to the list of target languages in directives:languages. The container will be named genapp_directives:application:menu:id-module:id unless directives:dockerhub has been defined as will be described in the next section.

The generated container images can be executed at the command line via "docker run genapp_directives:application:menu:id-module:id /genapp/bin/module:id input" where input is the JSON input object to the module's executable. The container will return the output object from the executable.

Once the container images are built they can be added to appconfig:resources for execution. For example to run on the local host with a web server user of "www-data", one could include in appconfig:[17]

```
,"docker-local" :
"docker run -v __rundir__:/genapp/run \
--user www-data \
genapp___application__:__menu:id__-\
__menu:modules:id__ \
/genapp/bin/__menu:modules:id__"
```

Note that the text prefixed and suffixed with double underscores will be dynamically modified with the appropriate values during module execution. The __rundir__ is a job specific directory which

[15] https://dibbs.seedme.org/

[16] https://genapp.rocks/learn under documentation → advanced topics → module dependencies

[17] Note the \'s are included here for formatting. In practice, the quoted value of :docker-local would be in one line.

is mounted to the container. This strategy could be extended by prefixing an ssh command to the above value of :resources:docker-local and creating a new resource, say :resources:docker-ssh, but one must take care to ensure an appropriate file system share if job input files are consumed or output files are produced by the executable.

To implement the "docker" target language required writing a target-language definition file and code fragments for the install script. This code is then assembled and run during the GenApp generation run. The generated code has access to the module definition data and proceeds by writing the appropriate Dockerfile from the information in the module file (module:dependencies, module:executable) for each module and subsequently builds the container images.

3.3 Abaco integration

Abaco has been integrated as a compute resource in GenApp. The docker images produced by the "docker" target language are Abaco compatible. Abaco runs the container with the input object placed in an environment variable. Abaco compatibility of GenApp containers is achieved by adding an entry to the Dockerfile describing the modules's container as follows:[18]

```
CMD /genapp/bin/module:executable `echo $MSG`
```

To run on Abaco, the image must be pushed to Dockerhub. This is enabled by adding a directives:dockerhub entry, e.g.:

```
"dockerhub" : {
  "id" : ""
  ,"user" : "docker hub user name"
}
```

The directives:dockerhub:user must be registered with Docker and the user must `docker login` before running GenApp for target language "docker". Once this is setup, the image will automatically be pushed to Dockerhub when GenApp generation is run. The directives:dockerhub:user/ will be prefixed to the generated docker image name, as well as the optional :dockerhub:id. Note the image name contains only the directives:dockerhub:user, directives:application and the optional directives:dockerhub:id. The module specific information is the the image tag. For example, a given directives:dockerhub:user of "xyz", directives:application of "demo", menu:modules of "simulate" and a menu:modules:simulate containing "energy" and "md", `docker image list` would return:

```
REPOSITORY TAG
xyz/genapp_demo simulate-energy
xyz/genapp_demo simulate-md
```

To run these images via GenApp on Abaco, one must first register with Abaco as described in its documentation[19]. The steps of creating a TACC account[20] and generating a token are currently required.[21] Once those steps are completed, one must add this information to secrets:abaco. For example:

```
"abaco" : {
  "host" : "https://api.tacc.utexas.edu"
  ,"username" : ""
  ,"password" : ""
  ,"api_key" : ""
  ,"api_secret" : ""
}
```

Where the values are filled in with the appropriate information. Finally, one can add the resource to appconfig:resources as follows:

```
,"abaco" : "abaco/abacorun"
```

The abaco resource runs the abaco/abacorun php wrapper which handles the API calls to secrets:abaco:host to register, execute, retrieve the results and delete the actor.

To implement Abaco execution required extending the "docker" target language code fragments to check directives:dockerhub. If directives:dockerhub is set, the container names are adjusted with the directives:dockerhub:user and id and the images are pushed to Dockerhub. Additionally, running on Abaco required writing a new program "abacorun" which takes the JSON input object from the command line, reads the secrets, makes a sequence of Abaco API calls to run the module's container, retrieve the results and finally write the JSON output object to standard output. The output object contains the results of the executable or error information if the run failed.

3.4 Demo site

A demonstration GenApp generated website is available[22]. Provided are examples of two modules: one, a simple "energy" calculator computing $E = mc^2$; the second, a test of the messaging system from the executable. There are three instances of each of these two modules: local execution; execution via a docker container; and execution via Abaco. For each of these execution methods, the respective module's underlying executable is identical. The website appears as shown in Fig. 4. To assist novice user navigation, tooltips are generally provided by hovering the cursor over the various buttons and inputs. The application's source code is available by clicking on the triple-bar menu icon at the top left and clicking on "Source".

To produce the demo site, we installed GenApp[23] on a Jetstream virtual machine. We started with the energy and message modules of the GenApp tutorial2 application and added the dependencies information as described in section 3.2. We then copied the energy.json module definition file to denergy.json and aenergy.json for container and Abaco execution respectively. To the denergy.json file we added resource:docker-local and to the aenergy.json file we added resource:abaco. The same steps were performed for the message.json file, producing modules amessage and dmessage. The six modules' ids (energy, message, denergy, dmessage, aenergy, amessage) were added to the menu.json file to allow them to appear to the user. We added appconfig:resources:docker-local and appconfig:resources:abaco as described in sections 3.2 and 3.3. The secrets.json file was created with abaco:host, :username, :password, :api_key and :api_secret values taken from our TACC and Abaco registration. directives:secrets was set to the path of our secrets.json file. directives:dockerhub:user was set to our DockerHub user. directives:languages was set to [html5, docker]. The GenApp command

[18] This is handled automatically by GenApp's "docker" target language.
[19] https://abaco.readthedocs.io/en/latest/getting-started/index.html
[20] Or elsewhere if so hosted.
[21] We will in future add a utility to GenApp to create this token.
[22] https://test.genapp.rocks/pearc19
[23] https://genapp.rocks/wiki/wiki/get

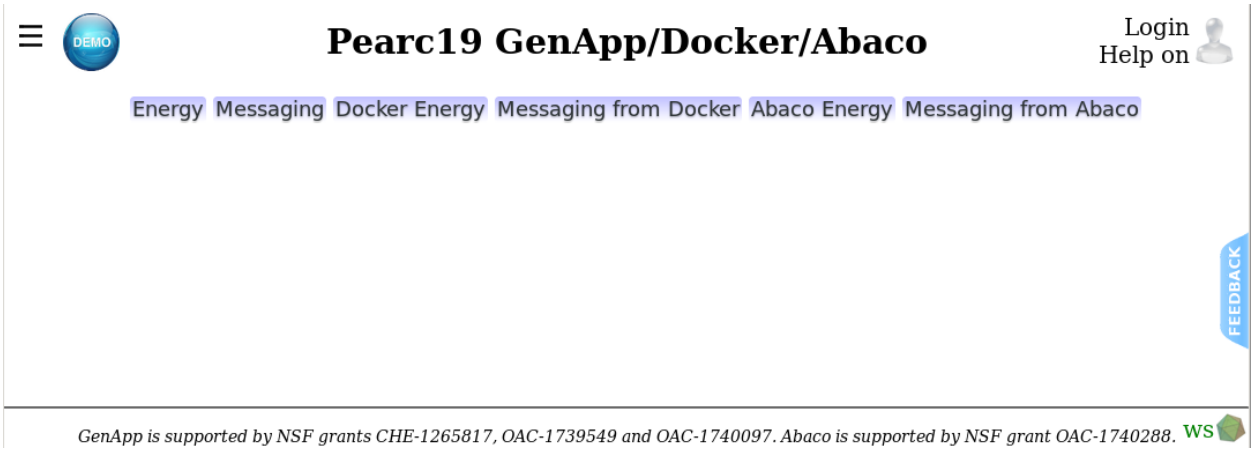

Figure 4: GenApp demo website. Example modules are provided for local, container and Abaco execution. The menu icon is at the top left. Clicking the menu icon will reveal the "Source" entry. The login and registration controls are at the top right. The Feedback tab is on the right. Note that login and registration are not required to run the examples. Figure © 2019 Emre Brookes

line program `genapp` was invoked which handled all the steps of building the website, the Docker container images and pushing them to DockerHub. The result of these steps is the working demo site.

4 USE CASES

Having module's executables containerized provides ease of execution portability. Our plan is to enhance existing GenApp application modules with dependencies to allow generation of docker container images, allowing new and existing deployments of applications to utilize container enabled resources. Abaco is one such container resource. Additionally, having the depencencies defined allows automation of installation of executables in traditional non-container resource environments.

5 FUTURE WORK

GenApp modules may require user file input and executable generated file output. Support for staging of input files and recovery of output files will be added. GenApp currently uses an Actor per job for Abaco submission. Long lived "stateless" Actors could process multiple jobs, reducing overhead for job startup and shutdown. Other container technologies exist besides Docker, for example Singularity containers[24] are often used in HPC environments. Given a use case, GenApp could be extended to optionally build Singularity containers.

6 CONCLUSIONS

Extending GenApp capabilities is a straightforward process. Definition file driven GenApp now supports container generation from defined modules in an Abaco compatible format which can optionally be pushed to DockerHub. Executable dependencies are added to the module definition file to enable containerization. Containerization simplifies transportation of executables across resources, enhancing ease of generated application deployment. Abaco resources support the Actor Model of concurrent computation and has been integrated as a resource with GenApp.

[24] https://singularity.lbl.gov

ACKNOWLEDGMENTS

This work is supported by the NSF grants CHE-1265817, OAC-1740097 and OAC-1912444 and NIH grant GM120600 to E. Brookes and NSF grant OAC-1740288 to J. Stubbs. We are grateful to application developers and their users for their valuable feedback and suggestions. This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number ACI-1548562.

REFERENCES

[1] Stubbs J., Vaughn M., and Looney J. 2018. Rapid Development of Scalable, Distributed Computation with Abaco. *Proceedings of the 10th International Workshop on Science Gateways* (2018).

[2] Emre Brookes. 2014. An open extensible multi-target application generation tool for simple rapid deployment of multi-scale scientific codes. In *Proceedings of the 2014 Annual Conference on Extreme Science and Engineering Discovery Environment*. ACM, 53. https://genapp.rocks

[3] Emre Brookes, Nadeem Anjum, Joseph Curtis, Suresh Marru, Raminder Singh, and Marlon Pierce. 2015. The GenApp framework integrated with Airavata for managed compute resource submissions. *Concurrency and Computation: Practice and Experience* 27, 16 (2015), 4292–4303.

[4] Alexey Savelyev and Emre Brookes. 2017. GenApp: Extensible tool for rapid generation of web and native GUI applications. *Future Generation Computer Systems* (2017). https://doi.org/10.1016/j.future.2017.09.069

[5] Stephen Perkins, David Wright, Hailiang Zhang, Emre Brookes, Jianhan Chen, Thomas Irving, Susan Krueger, David Barlow, Karen Edler, David Scott, and N. Terrill. 2016. Atomistic modelling of scattering data in the Collaborative Computational Project for Small Angle Scattering (CCP-SAS). *Journal of Applied Crystallography* 49, 6 (2016). http://ccpsas.org

[6] 2016. Jetstream, first NSF-supported cloud infrastructure for science & engineering research, to launch September 1. https://itnews.iu.edu/articles/2016/jetstream,-first-nsf-supported-cloud-infrastructure-for-science--engineering-research,-to-launch-september-1.php. (2016). Accessed: 2017-03-01.

[7] John Towns, Timothy Cockerill, Maytal Dahan, Ian Foster, Kelly Gaither, Andrew Grimshaw, Victor Hazlewood, Scott Lathrop, Dave Lifka, Gregory D. Peterson, Ralph Roskies, J. Ray Scott, and Nancy Wilkins-Diehr. 2014. XSEDE: Accelerating Scientific Discovery. *Computing in Science & Engineering* 16, 5 (2014), 62–74. https://doi.org/10.1109/MCSE.2014.80

[8] Suresh Marru, Lahiru Gunathilake, Chathura Herath, Patanachai Tangchaisin, Marlon Pierce, Chris Mattmann, Raminder Singh, Thilina Gunarathne, Eran Chinthaka, Ross Gardler, and A. Slominski. 2011. Apache airavata: a framework for distributed applications and computational workflows. In *Proceedings of the 2011 ACM workshop on Gateway computing environments*. ACM, 21–28.

[9] Omar Sefraoui, Mohammed Aissaoui, and Mohsine Eleuldj. 2012. OpenStack: Toward an Open-source Solution for Cloud Computing. *International Journal of Computer Applications* 55, 3 (October 2012), 38–42. Full text available.

[10] Emre Brookes and Alexey Savelyev. 2017. GenApp Integrated with OpenStack Supports Elastic Computing on Jetstream. In *Proceedings of the Practice and Experience in Advanced Research Computing 2017 on Sustainability, Success and Impact (PEARC17)*. ACM, New York, NY, USA, Article 11, 8 pages. https://doi.org/10.1145/3093338.3093356

[11] 2017. ECMAScript Language Specification. https://www.ecma-international.org/ecma-262/5.1/. (2017). Accessed: 2017-03-01.

[12] M. McLennan and R. Kennell. 2010. HUBzero: A Platform for Dissemination and Collaboration in Computational Science and Engineering. *Computing in Science Engineering* 12, 2 (March 2010), 48–53. https://doi.org/10.1109/MCSE.2010.41