The NEWT Platform: An Extensible Plugin Framework for Creating ReSTful HPC APIs

Shreyas Cholia
Lawrence Berkeley National Laboratory
scholia@lbl.gov

Terence Sun
Carnegie Mellon University &
Lawrence Berkeley National Laboratory
tsun@lbl.gov

ABSTRACT
This work describes the NEWT platform, a framework for creating ReSTful web APIs for high-performance scientific computing. The NEWT platform is designed to be a customizable framework that can be deployed at an HPC center, and enables access to various backend resources and services through a common web API. The goal of this effort is to create a service that can be plugged into multiple backend resources, and can easily be extended, while presenting a standard interface to the consumer with common semantics. This effort also updates the NEWT API that has been deployed at NERSC since 2010, and provides additional structure and consistency across the API.

Categories and Subject Descriptors
D.2.2 [**Software Engineering**]: Design Tools and Techniques – *Modules and interfaces*

Keywords
ReST, API, HTTP, web, HPC, Scientific Computing

1. INTRODUCTION
Science gateways, or web-based interfaces, for high performance scientific computing and storage systems enable new modes of scientific discovery through collaboration, and simplify the sharing of scientific products with the broader scientific community. At the National Energy Research Scientific Computing Center (NERSC), we engage with science teams interested in building these science gateways. In our work, we have noticed that many of the gateways involve bridging interactive front-end web interfaces with backend supercomputing and storage resources. Thus, the first iteration of NEWT was developed [5] to capture the major interactions between web applications and the resources at NERSC. This allowed web developers to build science-centric applications using their own tools and frameworks, while interacting with NERSC using a well-defined HTTP API.

However over time, we realized that the initial version of NEWT was primarily dependent on resources at NERSC, making extensibility across different platforms difficult. To address this issue, we have been developing a new version of NEWT with the focus of providing a well-defined HTTP API adaptable to any HPC center, built upon the base NEWT framework. The new version is built with modularity in mind, allowing developers to write modules, also known as "adapters", to provide NEWT access to any HPC resource.

We have now released this open-source implementation of the NEWT platform that is available to HPC centers that wish to deploy ReST APIs to expose their resources. A key part of this effort has been to enable a great degree of flexibility in allowing developers to extend and customize the service and the API to meet their own needs. In this paper we describe the general features of this new NEWT platform, the API, and the features that enable developers to adapt and expand NEWT to use multiple implementations of backend resources.

The NEWT platform can be downloaded at https://github.com/NERSC/newt-2.0/. We recommend some familiarity with the Python [21] programming language and the Django [8] web framework before working with the NEWT platform.

2. RESTFUL WEB APIS
The NEWT API tries to follow Representational State Transfer (or ReST) design principles as much as possible. ReSTful design helps define a common set of semantics across the API, mapping HTTP verbs, URLs and parameters to backend resources and their respective operations. For a detailed discussion of ReST and ReSTful web APIs see [1], [10] and [23].

For the NEWT API, we use the following HTTP methods to define the types of operations we want to perform on our backend resources. This roughly maps to the CRUD (Create, Read, Update, Delete) pattern for accessing resources [15]:

- POST: Create or Update a resource
- GET: Query a resource (Read-Only, Idempotent)
- PUT: Update a Resource
- DELETE: Remove a Resource

We use a hierarchical layout for our resources, so that parent resources provide a list of child resources contained by them, where applicable. Responses are standardized in a common envelope (described in section 3.2). Success or failure is encapsulated in the standard HTTP status codes [13] as part of the response.

3. NEWT API
3.1 API Components
The NEWT platform implements the ReSTful semantics described above on what we consider to be the typical set of

resources needed by web gateways and tools to interact with a scientific computing center. Specifically it exposes the following endpoints within the API:

- /auth/ - manage authenticated user access
- /account/ - user, group, accounting information
- /job/<host>/ - submit and query jobs on the designated host resource
- /command/<host> - run commands on the designated host resource
- /file/<host>/ - access files and directories on the designated host resource
- /store/ - user-defined database storage for application specific data
- /status/ - status information on resources exposed by other parts of the API.

Note that while we try to keep ReSTful principles in mind, there may be cases where we need to deviate from this style to meet specific technical needs. We direct readers to our previous work [5] and to the NEWT documentation [20] for more details on each of these components.

3.2 Standard API Envelope

In order for an API to be easily usable, it is important to maintain a standard response structure returned by it. By providing uniformity across different and potentially disparate elements of the API, clients are able to handle the response messages in a consistent manner. By default, NEWT encodes its responses using the JavaScript Object Notation (JSON) format, which is quickly becoming the de-facto standard for providing lightweight HTTP APIs.

In the NEWT API, we use the following envelope for all JSON responses:

```
{ 'status': 'OK'|'ERROR',
'status_code': <HTTP error code>,
'output': <API Response Output>,
'error': <Error Messages> }
```

Note that there are exceptions to the envelope on responses for obvious reasons. For example, retrieving files through NEWT's file API will not wrap the raw data in an envelope.

For developers, the NEWT platform also makes it easy by automatically wrapping responses and converting them to the standard NEWT envelope format. In other words, when implementing access for a new resource, the adapter does not need to adhere to the envelope format because the base code of NEWT will do it automatically.

4. NEWT PLATFORM ARCHITECTURE

The NEWT platform is developed using the Python [21] programming language using the Django web framework [8]. Django is a very popular framework for developing web applications and tools. Using the model-view-controller (MVC) model, Django provides for a clean separation between the interface, business logic and backend resources.

All top-level components of the NEWT API are implemented by a specific Django sub-module (known as an application). The application itself maps a set of URIs to specific functions or views. One of the significant improvements to the NEWT architecture is to make these view functions pluggable and modular. NEWT uses the adapter design pattern [12] to provide this modular behavior. The underlying functionality must be implemented by an adapter that connects the backend resource with the view function. Adapters are implemented as plugins, which allows for different adapters in different deployments of NEWT.

The specific adapter that implements an API component is defined in the NEWT configuration, using the Django "settings" module. The NEWT platform ships with an adapter template for each application that provides stubs for the basic functions and can be customized to match a system specific backend resource. Additionally NEWT also includes a reference implementation for each adapter (referred to as the local implementation) that can run on a standalone workstation with minimal external dependencies. We discuss this local implementation in Section 4.3.

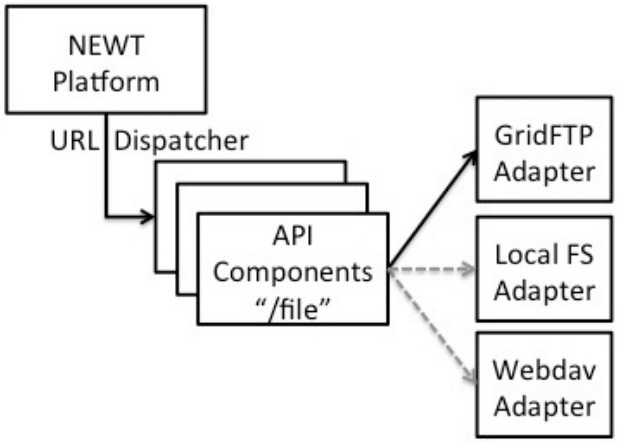

Figure 1. An example of the NEWT plugin architecture.

Figure 1. shows an example of this architecture. The NEWT platform dispatches requests for a specific URL to the appropriate API component application (e.g. /file). This interface is then implemented by a specific underlying adapter as determined by the settings for that deployment. Additionally the URL dispatcher can be extended to add and modify API components.

4.1 Adapters

In the new version of NEWT, the different modules are split into separate Django applications. Each application supports a set of URLs and maps those to a set of Python functions. However, by default these functions are null implementations and return an empty response with an HTTP "Not Implemented" status code

The actual API implementation is provided by adapters that can talk to the backend resource and return an appropriate response object. The NEWT code provides examples of multiple adapter modules for any given API component, which can be selected through a configuration file. For example, the file API has adapters for local file support as well as remote file access through GridFTP [3]. Similarly, the system status API has multiple adapters that can return system status by using the UNIX ping command or by querying an external HTTP URL.

There is also support for user-written adapters – in fact developers are encouraged to create their own adapters to match their own resources.

Each adapter has a template file that specifies the necessary functions of the adapter, thus allowing a developer to create an adapter for a specific purpose. For example, in the authentication adapter template, the developer would see the following function definitions: `get_status(), login(), logout()`.

In their adapter, the developer must implement the functions listed in the template as specified. Note that the adapter can simply return a Python object with the desired output. The NEWT framework will handle the aspects of wrapping the response in a standard JSON envelope.

4.2 Extensibility

While NEWT serves as a basis for fundamental HPC operations in numerous scenarios, it cannot provide function definitions in the template file for all use cases of an adapter. To allow for this functionality, every adapter includes an adapter-level URL router and a tuple list in the form of:

```
patterns = (
(url_regex, function, request_required),
...)
```

If the URL component following the base URL of the adapter is matched by the regular expression, it will call the associated function along with a Django request object (if specified). For example, here is the extension needed to return a specified image using a `get_image()` function:

```
patterns = (
(re.compile(r'/image/(?P<name>.+)$'), get_image,
False),
...
)
```

This extras router allows developers for NEWT to implement functionality outside of what is specified in the basic NEWT code and still maintains the level of abstraction NEWT was created for.

4.3 Local Implementation

As part of the downloadable package for NEWT, we offer a default "local" implementation of the NEWT API as an example of how an adapter can and should be structured. It is designed to allow developers to gain familiarity with the API on their local workstations, without any specialized resources connected to it. This also allows them to become familiar with the plugin approach, so that they can develop custom plugins to match their own HPC infrastructure.

The local implementation implements features in each of the adapters:

- /auth/ - Uses the default Django user authentication
- /account/ - Uses the default Django user model
- /job/ - Uses UNIX commands to simulate submitting a job
- /command/ - Uses the Python Popen module to run system commands
- /file/ - Uses the Python System module to access files on the local system
- /store/ - Uses Django's default sqlite3 database to store blobs of data
- /status/ - Uses the system ping command to get the status of the machine

Also included are 2 other user-store adapters that showcase the use of Redis [22] and MongoDB [16] as store backends. Both provide insight on how to implement other database technologies as backends.

5. MOTIVATING USE CASES

Much of the work on making the NEWT platform extensible was motivated by the following use cases:

5.1 Local Filesystem Access

The original NEWT code used grid tools to access remote filesystems. However, there can be a performance penalty associated with accessing a remote filesystem. In cases where the filesystem is mounted on the NEWT service host, it is more efficient to serve the files directly using local POSIX filesystem calls. In particular this was implemented for the Advanced Light Source SPOT portal [2], and allowed for a deployment of NEWT that allowed customized access control and local file access.

5.2 Alternate Database Backends

The original NEWT user store was implemented in MongoDB. MongoDB provides a very flexible database model and query interface for building custom user stores. However, it requires additional infrastructure that may not be available at a given site. By enabling a plugin model for NEWT, we allow developers to provide alternate implementations for a custom user store. We provide reference implementations that use MongoDB, Redis (for key-value based data), and SQL databases (using the built-in Django Object-Record-Model) to demonstrate the flexibility of this approach.

5.3 Site-specific Authentication

Most HPC sites tend to have very specific security requirements when it comes to authentication of users. Rather than prescribing a specific security infrastructure, it makes more sense to enable the NEWT platform to plug in to whatever authentication infrastructure already exists at a given site. For example, some sites may implement multi-factor authentication. Other sites may choose to delegate authentication to standard UNIX modules like PAM. In order for NEWT to be deployed in different domains, that may have different security requirements, we allow for callouts to custom authentication backends. This allows us to use a site-specific backend to make assertions about a given user.

NEWT authentication is built upon the standard Django authentication module, which supports several well-established authentication methods. This makes it possible to support almost any authentication scheme that is supported by Django, such as OAuth [14], Shibboleth [17] and basic HTTP authentication. Upon successful authentication, NEWT sends a token to the client in the form of a cookie, which is used to assert the user's identity in all subsequent communications.

5.4 Extensible Job and Account Information

Since different job managers and accounting systems expose different kinds of interfaces, we need to make sure that NEWT can talk to these different batch systems. At NERSC we support multiple job managers (Torque/MOAB, UGE), which means that NEWT needs to be able to maintain some level of extensibility in order to support them.

5.5 Current Gateways

The NEWT API has been in wide use at NERSC supporting several science gateways including the ALS SPOT Portal [2], The NERSC Online VASP Application [19], PDACS Galaxy [4] and the Dayabay Offline Data Monitor [6], among others. It is also the underlying platform for the MyNERSC User portal [18]. Our experiences with these projects have driven many of the features in building the new NEWT platform. It is expected that most of these gateways will be transitioned to the updated platform in the coming months.

6. RELATED PLATFORMS

Before concluding, we wanted to call out some other platforms that fill the space of ReSTful API development for scientific computing. In particular the Agave API Platform [9] developed by TACC provides an excellent alternative framework for building ReST APIs. While Agave takes a more full-featured approach out of the box, NEWT distinguishes itself by being a much more lightweight platform that allows developers to provide specific functionality through a plugin model that extend the default implementation. At the other end, there are more general tools like the Django REST Framework [7] or Flask [11], that make it easy to expose web APIs but do not capture HPC and scientific computing semantics as a core feature.

7. CONCLUSION AND FUTURE WORK

We hope that our work with the NEWT platform will make it significantly easier for scientific computing centers that are considering deploying their own web APIs for HPC. NEWT can help by providing them with the base framework for such an API, while offering a great deal of flexibility in being able to adapt and extend the code to meet their needs.

NEWT is intended to be open-source software (under the BSD license [24]), and we hope to receive contributions from the community in order to support other adapters for different types of functionality (e.g. other batch queue managers, alternate filesystem implementations etc.).

We also plan to investigate new features such as events and callbacks that would be more conducive to building complex job workflows. Another area of future development would be to support high-throughput computing, where clients would post units of work to an API channel, which would then be consumed by high-throughput workers.

We have had a great deal of success with our API driven approach in providing the building blocks to access a scientific computing center over the web. This has significantly enhanced and simplified science gateway development. We hope to see this approach successfully replicated at other sites as well.

8. ACKNOWLEDGMENTS

This research used resources of the National Energy Research Scientific Computing Center (NERSC), a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

9. REFERENCES

[1] Allamaraju, S. 2007-05. RESTful Web Services Cookbook (O'Reilly Media).

[2] ALS Spot Portal. http://spot.nersc.gov/

[3] Bresnahan, J., Link, M., Khanna, G., Imani, Z., Kettimuthu, R., and Foster, I. Globus GridFTP: What's New in 2007 (Invited Paper). Proceedings of the First International Conference on Networks for Grid Applications (GridNets 2007), Oct, 2007

[4] Chard, R., Sehrish, S., Rodriguez, A., Madduri, R., Uram, T.D., Paterno, M., Heitmann, K., Cholia, S., Kowalkowski, J., and Habib, S. PDACS - A Portal for Data Analysis Services for Cosmological Simulations.

[5] Cholia, S., Skinner, D. and Boverhof, J. NEWT: A RESTful service for building High Performance Computing web applications. Gateway Computing Environments Workshop (GCE), 2010. DOI= http://dx.doi.org/10.1109/GCE.2010.5676125

[6] Dayabay Offline Data Monitor. https://portal-auth.nersc.gov/dayabay/odm

[7] Django REST Framework. http://www.django-rest-framework.org/

[8] Django, http://www.djangoproject.com/.

[9] Dooley, R. et al. Software-as-a-Service: The iPlant Foundation API, 5th IEEE Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS). IEEE, 2012.

[10] Fielding, R. T. and Taylor, R. N. 2002. Principled design of the modern Web architecture. ACM Trans. Internet Technol. 2, 2 (May. 2002), 115-150. DOI= http://doi.acm.org/10.1145/514183.514185

[11] Flask. http://flask.pocoo.org

[12] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. 1995. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[13] HTTP: A protocol for networked information. http://www.w3.org/Protocols/HTTP/HTTP2.html

[14] Leiba, B. Oauth Web Authorization Protocol, Internet Computing, IEEE, vol. 16, no. 1, pp. 74-77, 2012.

[15] Martin, J. Managing the Data-base Environment. Englewood Cliffs, New Jersey: Prentice-Hall. p. 381. ISBN 0-135-50582-8.

[16] MongoDB. http://mongodb.org/

[17] Morgan, R.L., Cantor, S., Carmody, S., Hoehn, W., and Klingenstein, K. Federated Security: The Shibboleth Approach *EDUCAUSE Quarterly*, Vol. 27, No. 4. (0 2004), pp. 12-17

[18] MyNERSC. https://my.nersc.gov

[19] NERSC Online VASP Application. http://portal-auth.nersc.gov/nova

[20] NEWT reference documentation: https://newt.nersc.gov/

[21] Python Software Foundation. Python Language Reference, version 2.7. Available at http://www.python.org

[22] Redis. http://redis.io/

[23] Richardson, L. and Ruby, S. 2007-05. RESTful Web Services. (O'Reilly Media).

[24] The BSD 2-Clause License. http://opensource.org/licenses/BSD-2-Clause