NEWT: A RESTful Service for Building High Performance Computing Web Applications

Shreyas Cholia*
scholia@lbl.gov

David Skinner*
deskinner@lbl.gov

Joshua Boverhof*
jrboverhof@lbl.gov

*Lawrence Berkeley National Laboratory
Berkeley CA 94720, USA

***Abstract*—The NERSC Web Toolkit (NEWT) brings High Performance Computing (HPC) to the web through easy to write web applications. Our work seeks to make HPC resources more accessible and useful to scientists who are more comfortable with the web than they are with command line interfaces. The effort required to get a fully functioning web application is decreasing, thanks to Web 2.0 standards and protocols such as AJAX, HTML5, JSON and REST. We believe HPC can speak the same language as the web, by leveraging these technologies to interface with existing grid technologies. NEWT presents computational and data resources through simple transactions against URIs.**

In this paper we describe our approach to building web applications for science using a RESTful web service. We present the NEWT web service and describe how it can be used to access HPC resources in a web browser environment using AJAX and JSON. We discuss our REST API for NEWT, and address specific challenges in integrating a heterogeneous collection of backend resources under a single web service. We provide examples of client side applications that leverage NEWT to access resources directly in the web browser. The goal of this effort is to create a model whereby HPC becomes easily accessible through the web, allowing users to interact with their scientific computing, data and applications entirely through such web interfaces.

Keywords-science gateways; scientific computing; REST; HTTP; web; grid; AJAX; JSON; HPC

I. INTRODUCTION

Scientific computing is increasingly moving to the web. We have seen a strong demand for web access to scientific data and methods at the National Energy Research Scientific Computing (NERSC) Center, especially as web interfaces become more interactive and sophisticated. NERSC is the flagship high performance scientific computing facility for research sponsored by the U.S. Department of Energy Office of Science. NERSC is a national facility located at Lawrence Berkeley National Laboratory. As part of its mission to provide high quality resources and services that accelerate scientific discovery through computation, NERSC is helping to build a number of web gateways for science. In particular, we are developing the NERSC Web Toolkit (NEWT), a web service to facilitate the development of science gateways that can interact with High Performance Computing (HPC) resources.

NEWT is a collection of REST (Representational State Transfer) based services that allow scientists, programmers, staff, and the public to write web applications for HPC. This includes both anonymous read-only and authenticated read-write services designed to make HPC computing and data resources highly usable via a web browser. Web browser interaction is often mediated through scripts and web applications that poll, aggregate, and mash-up the content from the RESTful API's URIs. As such the API is not really a set of web pages, but a set of well defined URIs that web applications can invoke, which act as information sources and sinks.

We take a pragmatic approach to building HPC-to-web methods, acknowledging a restricted vocabulary of methods gathered from many years of watching how science teams use compute resources and data stores. The API is built around easy-to-use, concise, programmable interfaces. NEWT provides both open access for public data and LDAP/PKI authentication mechanisms for read-write management of jobs and data.

Our implementation of the NEWT API exposes the following resources as URIs that can be acted upon through standard HTTP verbs (GET, PUT, POST, DELETE):

- Authentication services
- Files and Directories
- Batch Jobs
- UNIX Shell Commands
- Accounting information
- Custom persistent stores
- System Status information

978-1-4244-9752-2/11/$26.00 ©2011 IEEE

NEWT requests consist of a combination of HTTP methods with URIs, parameters, and HTTP headers. NEWT responses are typically comprised of HTTP status codes, headers and JSON (JavaScript Object Notation) objects. We chose to encode our responses in JSON because this is the lingua franca of JavaScript enabled modern web browsers, simplifying the parsing and handling of these objects.

We believe that this method of interfacing with the HPC center will enable rich client side scientific computing applications that will be used for end-to-end user workflows entirely in the web-browser. While this paper focuses on the features described above, NEWT as an API is extensible. The REST architectural style lends itself to adding whole new categories of resources. NEWT can easily subsume other resources and APIs not mentioned here, as the scope of HPC expands.

In this paper we describe the motivations for moving to the NEWT model for building web gateways (section 2). We provide a summary of the REST architectural style (section 3) used by NEWT, along with an overview of current client-side technologies (section 4) that have made this model usable. We explain how this can be used for building gateways to HPC (section 5). We define the NEWT REST API at a high level (section 6) and delve into our implementation of the NEWT service (section 7). We briefly go over the performance of our implementation of NEWT (section 8). We guide you through writing a simple NEWT application (section 9), followed by example applications and use cases for NEWT (section 10). We describe some of the challenges we faced in building this service (section 11). We examine some of the related work (section 12), and future work (section 13) in this area before summarizing our conclusions (section 14).

II. RETHINKING WEB GATEWAYS FOR SCIENCE

Several scientific domains have benefited from web gateways. However, a large fraction of HPC still happens via command line sessions and shell scripts that run user jobs, post-process output, and perform analysis. We believe that this:

- excludes a large population of scientists that may not be comfortable with UNIX command line interfaces.
- results in one or two “specialists” that can run jobs on behalf of the rest of the science team.
- limits cross-domain science that may result from integrating well-exposed information from multiple sources.

There have been several attempts at building science gateways using specialized frameworks. While these simplify the process of building these gateways by providing some common interfaces and building blocks, they still require the need to learn a specific framework, and require programming from the “inside”.

Our goal with NEWT is to provide an interface that can be programmed by anyone with knowledge of HTML and JavaScript. Everything is a URI that can be acted upon through a standard HTTP command. All output is in JSON which is a native JavaScript object in the browser. *Asynchronous JavaScript and XML* (AJAX) and HTML5 technologies provide you with all the tools to interface with your HPC application. For example, in this model getting access to a dataset is simply a matter of entering a URI with appropriate parameters. This means that there is no need to understand the XML structure of an application framework or learn the semantics of grid credential management – NEWT handles all this automatically on the backend.

Prior to our work on NEWT, we have developed and deployed several web gateways at NERSC for various science communities. This includes custom standalone gateway applications (e.g. Deep Sky [15], QCD [14], GCRM) as well as generic portal based gateways (GridSphere [9]). In our experience, the customized, science-specific gateways tend to be much more useful, since they are built around the underlying science use-cases, rather than on the backend computational layer. The interfaces are more tightly integrated with the scientific applications themselves, thus allowing the users to focus on working directly with their jobs and data. On the other hand, we noticed that most of these gateways had several common patterns and methods for interacting with HPC centers. In order to facilitate maintainability and reuse of software, we started to focus on creating building blocks around these common access patterns such as user authentication, file transfers, job management and data sub-selection. Some of the generic portal technologies offered us common building blocks. But these were too closely linked to knowledge of grid access to the backend computational systems, and exposed several details that confused less experienced users. While these interfaces were customizable, there was a heavy learning curve and required application developers to create software interfaces within the frameworks themselves.

NEWT is based on the idea of encapsulating these building blocks in the form of HTTP URIs. The set of these URIs and the actions that can be performed on them give us an API. This enables science groups to build complete web gateways by using client-side technologies alone. The entire web-application is built around JavaScript/HTML/AJAX, which has a much lower barrier for entry.

III. REST

REST or Representational State Transfer [1, 25] is an architectural style commonly used for accessing resources on the web. The REST architectural style imposes a set of constraints on the application design including a client-server model, a stateless communication protocol, a uniform interface for all parts of the system, a layered resource hierarchy, and a cache-able set of responses. The HTTP protocol is the most commonly used implementation of the REST style (though the two aren’t necessarily the same thing). Since this paper focuses on a RESTful HTTP service, we would like to point out that when we discuss REST, we are specifically referring to REST as it applies to an HTTP service. However, it should be noted that REST is more general than HTTP and can be used by other protocols that meet a certain set of architectural constraints. Using the HTTP protocol to implement the REST pattern means that every resource can be represented as a unique URI and accessed through a set of HTTP verbs. This allows us to capture all the interactions needed to build an API to access these resources using REST.

A RESTful design constrains a web architecture for the purpose of simplifying usage, development, and deployment to the web. Firstly, this design requires the use of a client-server architecture that separates the user interface from the data storage concerns, and the biggest benefit is that development of components can proceed independently. The stateless constraint requires application state to be maintained exclusively by the client. Each message contains all the information necessary for the service to perform its function, and isn't reliant on a previous exchange. Responses must be marked as cacheable or not. The layered system requirement further enhances scalability, components are arranged in a hierarchy and each component's knowledge is restricted to the layer it is immediately interacting with. Lastly, the Uniform Interface constraint is fundamental to the design of any RESTful service - it requires all interactions between clients and services to be mediated through a few HTTP methods. The methods we concern ourselves with are GET, PUT, DELETE and POST. These methods are defined by the HTTP specification. A GET request is safe; it is not requesting any changes to be made to the server state. A PUT or DELETE request is idempotent, making multiple identical request will have the same result. Updating a resource multiple times with the same representation, or deleting a resource multiple times is equivalent to performing the operation once. A POST is a factory method that returns references to newly created resources, and is not safe.

Thus, the RESTful design constraints provide a standardized way to build an API using the HTTP protocol. This design includes correct use of standard HTTP methods and return codes. This is different from RPC protocols like SOAP, where HTTP merely acts as the envelope for RPC data and most information including the method is contained inside the XML payload and each service defines a new vocabulary.

IV. CLIENT TECHNOLOGIES

The advent of HTML5 and AJAX has turned the web browser into a sophisticated application development environment. HTML5 is a major revision of the HTML standard that adds several rich client-side features and interfaces into the web browser environment. A large subset of HTML5 is now supported in newer versions of most major web browsers [26]. JavaScript is now natively supported in modern web browsers, and AJAX allows for an asynchronous event based programming model, where the web client can make requests to the web server in the background while the rest of the application is running in the foreground. In particular modern browsers support the entire suite of HTTP verbs (HEAD, GET, POST, PUT, OPTIONS, DELETE) through AJAX making it possible to build RESTful applications entirely in JavaScript and HTML.

Additionally, there are several JavaScript libraries like JQuery, Prototype and YUI that provide easy-to-use abstraction layers to capture commonly used patterns. These libraries handle several low level details like cross-browser compatibility issues and protocol level AJAX interactions, allowing programmers to focus on the application logic. They also provide GUI widgets, effects and animations to help create a sophisticated front-end user experience.

This allows for a rich interactive platform - the web browser can now host fully featured graphical user interfaces that are very easy to use. Additionally there have been several visualization libraries (Protovis, Raphael etc.) that have been built on these JavaScript layers. These enhance the scientific computing workspace, by enabling advanced visualizations directly in the browser using data from remote web services.

These technologies have also enabled client-side applications bringing for mobile devices, opening up a whole new paradigm in terms of how scientific computing is accessed.

For our implementation of the NEWT service, we chose the JQuery and JQuery UI libraries [8] to provide users with helper functions that automate certain parts of the API. We found that JQuery allows us to express RESTful AJAX interactions in a terse and concise fashion. Additionally JQuery seems to have gained considerable traction in the web development community and features a large set of plugins and tools.

Since the NEWT service is primarily targeting JavaScript based web clients, the NEWT API uses JSON as its data interchange format. JSON has some distinct advantages [4, 5, 6]:

1. JSON is valid JavaScript, and every JSON object is automatically parsed by the browser as a native JavaScript object making it very easy to work with inside of JavaScript.
2. It is very lightweight - considerably more so than its primary competitor, XML.
3. It allows us to take advantage of JSON based NoSQL databases that provide us with a free-form persistent storage layer for web applications.

V. RESTFUL WEB SERVICES FOR HPC

The NEWT web service adheres to the REST style and implements a resource-oriented architecture (ROA). As differentiated from a RPC based web design where there are generally few URIs and most of the service logic is defined in the messaging layer, in RESTful ROA there are generally many, often infinite resources, each addressable through at least one URI and exposed through the Uniform Interface [2].

Generally in NEWT there are two types of resources for each component - a small set of one-off resources, some of which are factories for creating new resources via the POST method. These resources may contain an infinite number of sub-resources. For instance, a top-level filesystem resource can be considered a one-off resource, which may then contain an arbitrary nesting of sub-resources in the form of files and directories. Some sub-resources are also factory resources. A GET to a resource that contains sub-resources is a listing of some sort. When the POST method is used on a factory resource, the newly created resource's location will be returned to the client application. Idempotent methods PUT and DELETE are available on some sub-resources, but in many cases we allow for factory POST because the calling party may not want to be responsible for determining the final URI of the new resource. There are several top-level resource components: Authentication, Accounting Information, Job Management, File Management, Shell Commands, System Information and User-defined Persistent Storage.

A few operations allow anonymous access, but authentication is required for all operations that involve writing or utilizing restricted backend resources. NEWT's authentication component provides URIs for logging in and out. We use form-based authentication and send authorization data over secure transport (HTTPS) on privileged communications. In order to avoid authentication on every exchange we utilize sessions and cookies to maintain a little application state in the form of a session identifier on the server. This breaks the stateless constraint of RESTful design [3] but it is a common idiom on the web. Since cookies are natively supported and handled automatically by most modern web browsers, their usage allows users to build web applications and queries without having to explicitly manage authentication data with each request.

The account information component is read-only, and is used to access user profile information including group membership from the NERSC Identity Management (NIM) web service. The User-defined Persistent Storage component utilizes group membership to restrict read-write capabilities to "buckets" of information. This base URI of this component is a factory resource that allows users to create new buckets, and associate a bucket with predefined NIM groups allowing certain users to read and write data to that bucket. The representation of the base resource returned by GET is a listing of all the buckets. A GET against a bucket returns the entire contents of the bucket, which is a JSON dictionary. A bucket is also a factory resource and the POST method of a bucket will create a new document and return its location to the client application; PUT and DELETE are available for create/update and destroying the bucket. Each individual entry in a bucket is termed a document; PUT and DELETE can be used to create/update and destroy a document.

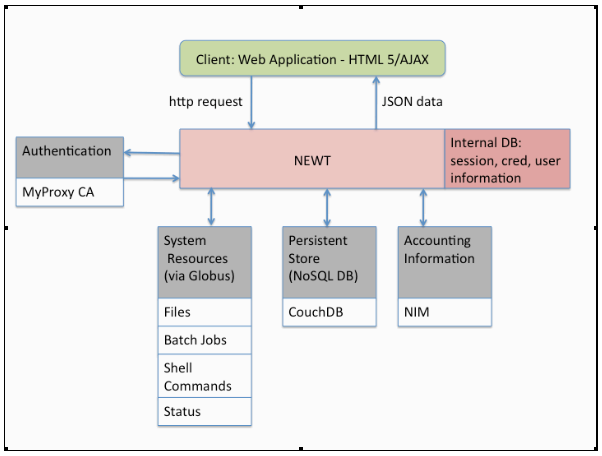

Client: Web Application - HTML 5/AJAX
http request
JSON data
Authentication
MyProxy CA
NEWT
Internal DB: session, cred, user information
System Resources (via Globus)
Files
Batch Jobs
Shell Commands
Status
Persistent Store (NoSQL DB)
CouchDB
Accounting Information
NIM

Figure 1. NEWT – RESTful Services for HPC

We use this general pattern of GET/POST/PUT/DELETE to access all resources that we are interested in. Fig. 1 provides an overview of the resources exposed by NEWT through its RESTful interface.

VI. NEWT API

In its current version, the NEWT API exposes the following resources and services:

A. *Authentication*

The authentication API allows users to log in with a username and password, check on the current login status, and log out. It relies on HTTP cookies to hold session state information. Internally, this automatically creates a short-lived grid certificate on behalf of the user that is tied to the session, which is used to access system resources as that user. However this grid-based interaction is completely hidden from the user and the API.

B. *Job Management*

The job management components of the API allow users to run jobs on the specified compute system. There are three modes of operation

- submitting a job to a batch queue and asynchronously querying the resource for job output.
- submitting a job to run immediately on a login node and asynchronously querying the resource for job output.
- running a job synchronously to get output and error streams.

There is also support for querying the list of all jobs in the batch queue. All the specifics of the batch scheduler (e.g. PBS or SGE job files) are hidden from the user.

C. *File Management*

The file management API allows users to download and upload files at a given URI. These are mapped to actual filesystem resources on the backend systems. The API also allows the ability to get filesystem directory listings as JSON objects.

D. *Accounting Information*

This is a read-only API that allows user applications to query accounting information from the NERSC Information Management (NIM) system. It allows us to expose objects in the NIM database as JSON data. This provide access to accounting and bookkeeping information such as user account information, quotas, hours used on a given system, group repository information etc. The API is flexible enough that it can transparently accommodate any new tables in that are created in the underlying NIM database.

E. *System Status Information*

This component provides a simple but important function - it displays whether a given system is available or not, thus providing applications with a sanity check before trying to access resources on the system. This is a public API and does not require authentication.

F. *User-defined persistent storage*

The user defined storage layer allows applications to store persistent data in the form of a JSON object. The store is completely free form, and can hold any valid JSON object. This provides a persistence layer that allows web applications to keep stateful information across multiple users or sessions.

The NEWT API takes the following form:

HTTP-verb base_url/top_level_resource/sub_resource

HTTP headers and parameters are used to pass in session and additional scoping information.

For example to get a directory listing on /tmp from a machine called "carver" through the REST API, we would need to invoke:

GET https://portal.nersc.gov/newt/system/carver/file/tmp/

where:

- base_url=https://portal-auth.nersc.gov/newt/
- top_level_resource =system/carver/file/
- sub_resource= tmp/

Some REST-like APIs will map the four HTTP verbs to the Create/Read/Update/Destroy operations. While this works for some use-cases, it can break certain RESTful constraints in more complex examples. Instead, we use the more general RESTful semantics:

POST - Write operation that alters or creates a new resource each time it is called. i.e. Multiple identical POSTs will each alter the state of the system or resource.

GET - Read operation that does not alter resource. i.e. Multiple identical GETs will return the same resource representation.

PUT – Write operation that creates/alters a resource once. i.e. multiple identical PUTs have the same effect as a single PUT.

DELETE – Delete operation that destroys a resource once. i.e. multiple identical DELETEs have the same effect as a single DELETE.

We have tried to stick to these semantics as closely as possible, with the understanding that some resource types may not fit into this model, and that we may diverge from the prescribed usage in order to simplify client-side programming. Using this combination of verbs and URIs we are able to come up with a concise API that enables us to manage HPC resources.

TABLE I. NEWT REST API OVERVIEW

Resource	**GET**	**POST**	**PUT**	**DELETE**
Files	Download file	Upload file	-	Delete file
Authentication	Get authentication status	Log in to NEWT	-	Log out from NEWT
Directories	Get directory listing	-	-	-
Shell Commands	-	Run a command	-	-
Jobs	Get job status/output	Create a job	-	Delete a job

Job Queue	Get batch queue	-	-	-
System Status	Get system status	-	-	-
Accounting Info (NIM)	Get information about a NIM resource	-	-	-
Persistent Store DB	Get existing documents in store	Create new store	Update documents in store	Delete store
Persistent Store doc	Get JSON document	Create new JSON document	Create or Update JSON document	Delete JSON document

Table 1. captures most of the actions supported by the NEWT API at a high level. We use a combination of HTTP Status codes, HTTP Response Headers and JSON data to handle the responses

The latest version of the NEWT API is available at https://newt.nersc.gov [17]. There are several subtleties and details in terms of URI construction and additional headers and parameters that need to be passed in that are described in the full API documentation.

VII. IMPLEMENTATION

The NEWT components interface with a variety of backend resources including grid services, NoSQL, the private NERSC Identity Management (NIM) web service and other web service APIs. Generally each component serves as a front-end to one backend resource, and components frequently communicate through a decoupled signaling mechanism. Grid credentials are generated during the login process, which are used for access to backend resources.

NEWT is currently implemented as a web service using the Python Django framework [20]. Django offers us a flexible web framework based on the Model-View-Controller pattern, which allows us a clean separation between the data models, business logic and front-end. It also has a powerful middleware stack that allows us to easily integrate authentication, user sessions and header manipulation into our application. Django does much of the heavy lifting when it comes to URI routing and request-response handling, allowing us to focus on the backend services that need to be exposed via the NEWT API. Also, since Django applications are written in python, they give us access to the rich eco-system of python based tools and libraries for scientific computation.

Since many of the resources served up by NEWT require authorized access, authentication plays a key role in the NEWT service. Authentication assertions are provided by an internal MyProxy service [22], which verifies authentication data against an LDAP database. This service returns a short-lived grid credential that is cached by the NEWT service for the lifetime of the user session. This credential is used for accessing compute and data resources that are exposed via the Globus grid toolkit [23]. The Globus Gatekeeper and GridFTP service interfaces are installed on all file and job resources that are made available via NEWT. They provide remote execution, batch and file management capabilities and allow us to perform actions as the actual user making the request. The Globus services map the grid credentials to local accounts that are then used to access the underlying resources.

However, the user never deals with any of the grid semantics - the NEWT service handles all the credential management and resource delegation. From the users perspective they are dealing directly with the underlying resource. We have deliberately avoided exposing grid interfaces in the NEWT front-end since they create an additional layer of complexity. The goal of the application is to deal with the resources directly. The NEWT web service hides the details that handle this interaction. In our case, we found the Globus toolkit provides us with a convenient execution and file transfer engine that preserves user authorization. However, this is purely an implementation detail and should not be exposed to the user.

NEWT also provides a RESTful interface to a user-defined persistent store. This allows web applications to store stateful information. This is currently implemented using CouchDB [24] - a NoSQL database that can store JSON data. CouchDB itself has a RESTful interface and deals directly with JSON data, making it very easy to translate NEWT requests into CouchDB requests. Web applications store free form JSON data in the persistent store and CouchDB is natural fit for supporting this functionality.

Additionally NEWT also interfaces with other Web APIs on the backend. The NIM account management system provides accounting and banking information for NERSC users, scientific collaborations and HPC systems. There are also other web-based tools that display system information. NEWT allows us to capture various disparate backend web APIs under a single umbrella, so that they are exposed through a single interface.

The NEWT web service is only available as an SSL HTTPS service. All transmissions are encrypted between the browser and the web server – no information is sent in clear-text over the wire. This prevents session or password hijacking on open networks, and ensures data integrity.

Session Cookies are restricted to the domain of the NEWT service itself. Additionally, sessions lifetimes are bound to the lifetime of the backend MyProxy credential and are no longer valid, after the lifetime of the credential.

VIII. PERFORMANCE

We consider the value of the NEWT API and service to lie in its ability to abstract resources and to provide ease-of-use to web application developers building interfaces to scientific computing. However, it is important to ensure that the addition of such an abstraction layer does not result in a significant performance penalty that would deteriorate the overall user experience.

Fig. 2 summarizes the NEWT performance and overhead, based on the current implementation of the NEWT service.

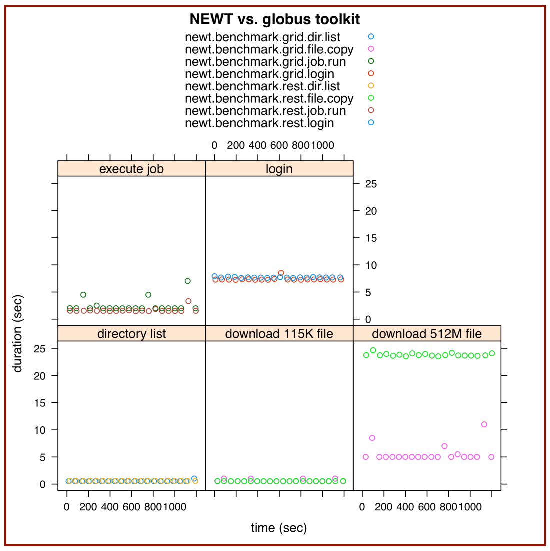


Figure 2. NEWT Performance and Overhead

We found that there is very little overhead for most standard Grid operations in the NEWT layer. The overhead is on the order of 0.05 to 1 seconds. NEWT is only involved in the request/response processing and has a small database overhead. A bulk of the time is spent inside the Globus layer itself. This includes authentication, job submission and small file transfer.

However, for file upload and download operations we perform a stage and copy. For large files this adds additional cost to the operation because the file has to be copied between the client machine, the NEWT service and the backend filesystem resource. For a 512 MB file this overhead was on the order of 2.5-3 to times the cost of a simple GridFTP copy.

It should be noted that the NEWT service is not currently meant to address the high performance requirements of large file transfers and bulk data movement – the focus is on building easy to use web interfaces where the web application itself consumes a small subset of the data. We offer other alternatives for these types of data needs including an HTTPD bridge that can directly access the global filesystem at NERSC and high performance data transfer nodes that run dedicated GridFTP services. In future versions we plan to look into more advanced techniques like streaming the data directly to the client, instead of staging it to disk.

Also, to cut down on Django overhead, we have configured the Apache/WSGI/Django interface to run in daemon mode, which allows the underlying application to run without a startup overhead. The python application is preloaded into the WSGI container and the application is very responsive to user requests.

IX. WRITING A NEWT APPLICATION

In this section we walk through a simple web application implemented by the NEWT API. This application will authenticate the user, check the status of a system, run a command and retrieve the output.

Note that steps 1 and 2 are automatically handled by a JavaScript library that we provide called "newt.js". This library is typically included with every NEWT application.

1. Get the current authentication status

GET /login/getuser/

Output: STATUS 200,

{"username": null, "auth": "false"}

2. Log the user in

POST /login

Params: username=myuser, password=mypass

Output: STATUS 200,

{"username": "myuser", "auth": "true"}

At this point the user is logged in and session information is stored in a cookie. This is handled by the web browser.

3. Get the status of a given host

GET /system/myhost/status

Output: STATUS 200,

{ "status": "UP", "machine": "myhost" }

4. Execute a job

POST /system/myhost/jobmanager/run

Params:

executable="/path/to/my/binary params"

Output: STATUS 200,

{"output": "DONE", "error": ""}

5. Retrieve output file

GET /system/myhost/file/path/outputfile?view=read

Output: STATUS 200, Binary file data

6. Save information in store

PUT /store/dbname/testdoc

Params:

{"run": 1, "output": "mydata"}

Output: STATUS 200

In the above examples a status code other than 200 indicates an error and requires error handling.

Any of the methods in this interchange can be succinctly expressed by the JQuery $.ajax function.

For example step 5. can be written as:

```
$.ajax(
url: newt_base_url +
"/system/file/path/outputfile?view=read",
type: "GET",
success: function(data) {
  // data has the contents of outputfile
 },
error: function(data) {
 // handle the error
 }
});
```

The above example demonstrates how a web application can be built to interact with the HPC center through a RESTful API over AJAX.

X. SAMPLE USE CASES

While NEWT is still in the development phase we are actively engaging with science teams to start building applications using this API. We have built some example applications to demonstrate the usefulness and functionality of NEWT.

1) Use Case 1 – Filesystem Treemap

The NEWT based Treemap application is motivated by use cases from to the STAR Virtual Organization. Many of these users have scattered datasets stored through their directories and are unfamiliar with the available command line data management tools. They need a simple visual way to analyze their disk usage and to trigger actions on specific files and directories.

Fig. 3 demonstrates a Treemap application that visualizes filesystem usage using NEWT and Protovis. This tool allows users to quickly assess the space usage for various files and directories, along with the ability to drill down into a particular directory. NEWT allows the web client to pull in directory usage information, which is then rendered using Protovis to provide an intuitive

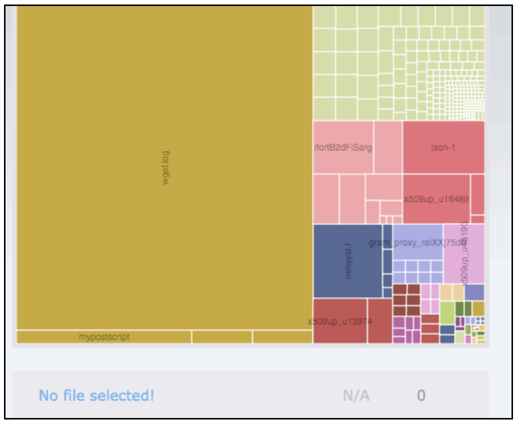


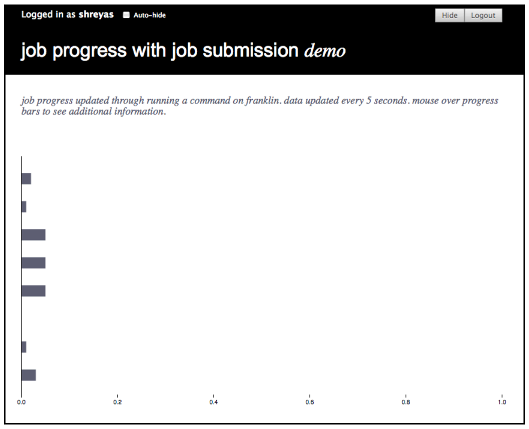
Figure 3. Treemap of Filesystem using NEWT and Protovis

2) Use Case 2 – Job Progress Counters

A common usage scenario for many-task jobs includes the ability to monitor the various tasks associated with a job. For a large distributed job, monitoring overall progress is much easier when the state of any given process can be graphed. For jobs that use emit performance or status counters periodically, we can consume this data and render it in a client browser.

Assuming that the job tasks emit these counters in a well-defined location and structure, NEWT allows for a simple interface to trigger a job submission, periodically polling for the current set of job counters, and informing the user when the task is done.

Fig. 4 demonstrates a job monitoring application for a code running on the NERSC Franklin system. The job periodically emits counters as JSON data, which are fed back to NEWT and displayed as progress bars in the client browser.



job progress with job submission *demo*

job progress updated through running a command on franklin. data updated every 5 seconds. mouse over progress bars to see additional information.

Figure 4. Job Progress Counters through NEWT

3) *Use Case 3 – VASP*

The Vienna Ab-initio Simulation (VASP) is the most widely used HPC application at the NERSC center. In order to facilitate users that may not be HPC savvy we wish to create a web client application in NEWT that can help generate batch files and submit jobs for VASP so that the user need not use Unix command line tools. It will also minimize the need to prepare VASP input files by providing graphical controls for specifying settings

We are currently gathering requirements and helping develop early use cases for several real world science projects, including data management tools for the fusion physics community, a web portal for the VASP application, a mobile queue status application, and sub-selection interfaces for NETCDF climate data. The next steps for NEWT involve community involvement in crafting a set of applications which can help inform the NEWT developers what aspects of the API can be improved or extended to reinforce the goals of making HPC application development on the web easy and productive.

Some motivation for what applications are likely to be useful for existing HPC user communities come from current web offerings in the form of batch script generators for applications like GAUSSIAN, VASP, and WIEN2K. This web forms produce the text suitable for submission, at the command-line, to a batch queue. With NEWT we can close the loop somewhat more simply, and rather than a copy-paste to a file via the command line for submission, the web-form itself can execute the submission. Other than including the “newt.js” file and ensuring authentication takes place, very little changes between the batch script generator and a complete application portal. Similar tasks for retrieval and analysis of data artifacts from the job may or may not be required.

XI. CHALLENGES

While the newer asynchronous JavaScript technologies have allowed developers to create rich web applications, they have also introduced avenues for security exploits. In particular, cross-site scripting (XSS) vulnerabilities have created unwanted leaks, where protected data is posted to or requested by malicious sites. In order to prevent XSS vulnerabilities, modern browsers have imposed limitations where a web page can only make AJAX calls to a service hosted in its own domain.

However, there are genuine use cases where cross-site access is desirable. In particular RESTful web APIs would be restricted to pages on a single domain, limiting their usefulness and availability. In developing the NEWT service we found a couple of techniques to manage cross-site access in a secure fashion:

Hosting a secure proxy on the server that hosts the web application. The client makes calls to its local proxy. This proxy forwards all NEWT requests to the NEWT service and passes responses back to the client

Using a newly emerging W3C standard called Cross-Origin Resource Sharing (CORS) [7, 10]. CORS allows for cross-site access in a secure and controlled way.

Using CORS, the remote web service itself can specify whether it will accept cross-origin requests through a set of HTTP headers (Access-Control-Allow-Origin, Access-Control-Allow-Methods, Access-Control-Allow-Credentials) and can restrict the domain for which an HTTP cookie is valid .

Additionally the client must explicitly pass along credential information in the AJAX XMLHttpRequest (XHR) object being used to make the request by setting:

XHR.withCredential=true

In this way cross-site access happens through an explicit handshake, and can be restricted to desired clients and servers.

There are also places in the API where we had to deviate from a pure REST implementation.

We do not use a strict REST based authorization scheme, and rely on cookie-based sessions instead. A purely RESTful implementation would require passing a username and password with every request, or explicitly including session information in the URI itself. Both of these techniques are cumbersome and make the API harder to use. Cookies and sessions, on the other hand, are handled automatically by web browsers. Since we consider browsers to be the primary target of NEWT, we made this decision to reduce the developer burden of managing authentication state information. We also provide a convenient JavaScript library that creates a login toolbar, and handles the authentication and authorization.

AJAX has trouble handling redirects, and will often fail silently. When passing back stateful information, like the location of a newly created resource, many RESTful APIs will attempt to redirect the client to the resource itself. Since this does not work well with AJAX, we do not use redirects - instead we pass along the required information as part of the JSON object in the body of the response. This also makes it much easier for an AJAX client to parse this information.

Given the push to make the browser do much of the heavy lifting in web application development, it can be easy to overdo this. We ran into issues where applications pulled in extremely large datasets and caused the web browser to run out of memory. It should be noted that applications built using interfaces like NEWT must be designed to account for these kinds of issues. Application developers should have a clear understanding of what tasks are best performed in the browser, and what tasks are run in the HPC environment. NEWT merely serves as an interface between the two.

XII. RELATED WORK

The OGCE community has developed and curated numerous science gateways and frameworks for enabling

HPC. Significant efforts include the Gridsphere [9] and OGCE portals [29] which allows for interaction with Grid resources through a portal framework. Our work takes a slightly approach – rather than building applications inside a specific framework, we push all application logic to the client itself. This allows application developers to build their entire application in client-space and gives them complete control over the user environment. Communication with the server happens completely over the RESTful API.

More recently there have been efforts that have taken a RESTful approach to interacting with Grid Services. The OGCE OpenSocial gadgets [30] allow developers to build web applications that access HPC as OpenSocial gadgets that can be hosted in containers like iGoogle. NEWT takes a slightly different approach to this problem – there is no intermediate OpenSocial layer and the client interacts directly with a variety of resources through a single HTTP interface. Rather than offering a collection of OpenSocial gadgets, NEWT integrates a heterogenous set of resources under a common API.

Other efforts overtly expose grid resources using REST. For example the Globus.org service has a RESTful API [27] that enables interaction with the grid resources through HTTP. Ian Stokes Rhees work "A REST Model for High Throughput Scheduling in Computational Grids" [28] describes a model for using REST to interface with Condor based Class-Ads and Grid schedulers.

We believe that NEWT is uniquely positioned in that the API encompasses both Grid and non-Grid resources through a RESTful interface. The API itself completely hides all details of the grid from the client application - the implementation semantics are never exposed. This means that the user simply interacts with resources directly. This allows for a mixed set of HPC resources the can be accessed through a uniform interface. In our implementation NEWT combines non-grid resources such as NIM or CouchDB with grid resouces in a seamless manner. And while the Globus Toolkit provides us with an implementation layer for accessing HPC resources, we can completely change this underlying implementation layer without affecting the client accessing the resource. For example, we can replace the globus layer with a localized login, filesystem-based data access and a direct interface to the batch scheduler without altering the client semantics.

XIII. FUTURE WORK

As mentioned earlier, we are currently working with science teams to build web-driven applications for scientific computing. We expect these efforts to feed into the NEWT API, and to drive many of the changes and new features in the service.

We expect to grow this set of services to include other resources as we move forward. Some of the proposed extensions to the API include

A layer that can access the globus.org service to move data across the wide area network.

A RESTful interface for Integrated Performance Monitoring of MPI jobs

Data sub-selection interfaces similar to OpenDAP

Integrating other existing RESTful APIs. In particular RESTful Cloud APIs will allow NEWT applications to integrate with existing cloud computing infrastructures.

There is also interest in being able to integrate authentication and authorization with common federated identity management solutions like OpenID, OAuth and Shibboleth/SAML. This will allow us to move away from password-based authentication towards a single sign-on type solution.

NEWT as an API is very extensible, because REST makes it easy to add resources to the API structure. NEWT is not simply restricted to grid services – we can expose a number of backend services as a web API. We already have examples of NEWT integrating with other RESTful APIs including CouchDB and NIM, and hope to extend this list as more HPC resources present their own RESTful APIs. In this context NEWT would serve as an umbrella for multiple grid, cloud, database and other HPC resources that would become intergrated under a single interface.

XIV. CONCLUSIONS

We believe that in the near future, the web will drive much of Scientific Computing. The NEWT service seeks to make High Performance Scientific Computing space more friendly to the web, by providing a RESTful API that interfaces well rich interactive web applications. While NEWT was specifically written with the NERSC center in mind, we have tried to make many of the high level constructs agnostic to the backend interfaces. As more and more services expose Web APIs, it becomes possible to consume and interact with data from multiple sources, enabling web "mash-ups" or applications that engage multiple backend web resources and APIs. This has the potential to create new and unique ways of exploring science and to bridge disparate scientific datasets and domains. For instance, we already support an OpenDAP web service at NERSC for NETCDF data sub-selection. A service like NEWT allows us to run a simulation that would help determine the sub-selection parameters. This workflow can now be combined under a single web application that makes HTTP requests using AJAX.

By describing the NEWT service, we hope to provide a model for building RESTful interfaces to HPC, and to stimulate research and development in other platforms that expose such Web APIs. . A long-term goal of this work lies beyond making HPC web applications easy to write and toward make them portable. In particular a web application written against NEWT, given appropriate backend implementations at multiple HPC centers, should be trivially portable. We look forward to testing such portability and ease-of-use aspects in the future.

ACKNOWLEDGMENTS

This work was funded in part by the Advanced Scientific Computing Research (ASCR) in the DOE Office of Science under contract number DE-C02-05CH11231.

REFERENCES

[1] 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.

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

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

[4] Crockford, D. 2006. The application/json Media Type for JavaScript Object Notation (JSON). RFC 4627. http://www.ietf.org/rfc/rfc4627.txt.

[5] Introducing JSON. http://www.json.org/.

[6] Crockford, D. 2008. JavaScript: The Good Parts (O'Reilly Media).

[7] Van Kesteren, A. 2010. Cross-Origin Resource Sharing – W3C Working Draft 27. http://www.w3.org/TR/cors/.

[8] JQuery Documentation. http://docs.jquery.com.

[9] Novotny, J., Russell, M., and Wehrens, O. 2004. GridSphere: An Advanced Portal Framework. In *Proceedings of the 30th EUROMICRO Conference* (August 31 - September 03, 2004). EUROMICRO. IEEE Computer Society, Washington, DC, 412-419. DOI= http://dx.doi.org/10.1109/EUROMICRO.2004.33.

[10] Ranganathan, A. CORS in Action. http://arunranga.com/examples/access-control/.

[11] HTTP Access Control. Mozilla Developer Network. https://developer.mozilla.org/en/HTTP_access_control.

[12] Bostock, M. and Heer, J. 2009. Protovis: A Graphical Toolkit for Visualization. *IEEE Transactions on Visualization and Computer Graphics* 15, 6 (Nov. 2009), 1121-1128. DOI= http://dx.doi.org/10.1109/TVCG.2009.174.

[13] Protovis. http://vis.stanford.edu/protovis/.

[14] The Gauge Connection. http://qcd.nersc.gov.

[15] Deep Sky. http://deepskyproject.org.

[16] Law, N. et al. 2009. The Palomar Transient Factory: System Overview, Performance and First Results. In *Instrumentation and Methods for Astrophysics*.

[17] NEWT API. https://newt.nersc.gov.

[18] Cornillon, P., Gallagher, J., and T. Sgouros. 2003. OPeNDAP: Accessing Data in a Distributed, Heterogeneous Environment. In *Data Science Journal*.

[19] Shneiderman, B. 1998. Treemaps for space-constrained visualization of hierarchies. http://www.cs.umd.edu/hcil/treemap-history/.

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

[21] Chan, S. and Andrews, M. 2006. Simplifying Public Key Credential Management Through Online Certificate Authorities and PAM. In *Proceedings of the 5th Annual PKI R&D Workshop*, April 2006.

[22] Basney, J., Humphrey, M., and Welch, V. 2005. The MyProxy online credential repository: Research Articles. *Softw. Pract. Exper.* 35, 9 (Jul. 2005), 801-816. DOI= http://dx.doi.org/10.1002/spe.v35:9.

[23] Foster, I. Globus Toolkit Version 4: Software for Service-Oriented Systems. IFIP International Conference on Network and Parallel Computing, Springer-Verlag LNCS 3779, pp 2-13, 2006.

[24] The CouchDB Project, http://couchdb.apache.org/.

[25] Representational State Transfer (REST). http://www.packetizer.com/ws/rest.html.

[26] HTML 5 and CSS3 support. http://www.findmebyip.com/litmus.

[27] REST API Documentation – Globus.org. http://confluence.globus.org/display/GOPUB/REST+API+Documentation

[28] Stokes Rees, I. 2006. A REST Model for High Throughput Scheduling in Computational Grids. Oxford Univ., - 248 p.

[29] Open Grid Computing Environments. June 2009. Pierce, M., Marru, S., Wu, W., Kandaswamy, G., von Laszewski, G., Dooley, R., Dahan, M., Wilkins-Diehr, N., and Thomas, M. TeraGrid 2009, Arlington VA.

[30] OGCE Gadget Container. http://www.collab-ogce.org/ogce/index.php/OGCE_Gadget_Container