Mem. S.A.It. Vol. 80, 365

Memorie della

A RESTful catalog for simulations

R. Wagner

Center for Astrophysics and Space Sciences, University of California at San Diego, La Jolla, CA 92093, e-mail: `rwagner@physics.ucsd.edu`

Abstract. To support describing data generated by our group and in the Computational Astrophysics Data Analysis Center (CADAC), I have designed a web service based on Representational State Transfer (REST) to catalog the simulations, data and software. This catalog (in particular its web service interface) was designed with three goals in mind: To capture sufficient detail to enable publishing to the VO as theory standards emerge; be simple enough to non-web developers to write client code for; and be usable on a day-to-day basis for tracking simulations as they evolve by getting the data into the system as it's being produced. I will present our motivation for this approach, provide a introduction REST by designing an example catalog, and include a minimal introduction to SimCat, the full simulation catalog.

Key words. Catalogs - Methods: numerical

1. Introduction

This paper describes a web service developed by our group, the Laboratory for Computational Astrophysics (LCA), to catalog simulations. The service was designed using an architectural style called *Representational State Transfer*, or REST. (An introduction to REST and my motivation for choosing it over other styles will be covered later.) I will not go in to much detail on the implementation, but will instead concentrate on the service interface, since this is where most of the important design decisions were made. Particular emphasis will be placed on the choice of REST based on the needs of the users.

Send offprint requests to: R. Wagner

1.1. Motivation

As can be guessed from the name of our group, our primary focus is performing simulation of astrophysics. We also maintain two public codes, Zeus (a computational fluid dynamics and radiation-hydrodynamics code) and Enzo (a hybrid hydro adaptive mesh refinement code). Both of these codes are parallel applications, and Enzo in particular scales to over 4096 processors (see Fig. 1), and this trend will likely continue. One side effect of this performances is the need to manage incredibly large datasets. A single simulation can range from 100 GB to 100 TB, and our current archive exceeds 500 TB. Another side effect of computing at this scale is the difficulty in repeating simulations that require millions of CPU hours. As a result, we take much care in scaling up the simulations and testing new physics to ensure both an accurate and reliable program,

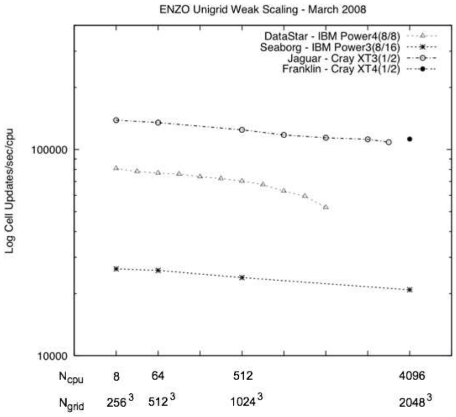

Fig. 1. Weak scaling study for an Enzo unigrid simulation. Flat lines indicate a constant time to solution as both the problem size and number of tasks are scaled together.

and top performance during the full-scale run. These issues have made managing and tracking simulations and their data a top priority for us.

The need for organizing simulations is hardly limited to our group, and one result of our need for better data handling is the Computational Astrophysics Data Analysis Center (CADAC)[1], organized by. This was organized to support a workshop on star formation at KITP, which included a comparison of turbulence simulations from various codes. The CADAC brings a community of users together with compute resources at SDSC, and a data grid, using the SRB (also hosted at SDSC). During the workshop, it was sufficient for users to work with each other to manage the simulation results for comparison. But looking ahead, we would like integrate this data into a public archive that supplements publications. In fact, in addition to the workshop results, our group has contributed the Simulated Cluster Archive, a series of galaxy cluster simulations using different physics models. These datasets (the combination of the turbulence and galaxy cluster simulations) are a great starting point for a more general computational astrophysics archive, and the ability to describe this data helps to define our needs.

This description of the LCA and the CADAC is just a snapshot in time. The data keeps being produced, and while we may be able to find the files or directories, over time we lose track of what's in them. While we may be able to recover some information about the contents, such as a parameter file, we quickly lose track of other information, such as what version of the software was used to produce the data. This has led to the design of a system that can capture as much information as possible, as early as possible.

1.2. IVOA Theory Interest Group

In the present day, it's useful to consider any existing standards when designing a piece of software, particularly a web service. Things like common data formats and shared interface can greatly improve the sharing of results. The International Virtual Observatory Alliance (IVOA) standards focus on web services and data models, with associated XML Schemas.

While there are no theory-specific standards (as of today), the Theory Interest Group of the IVOA has been working towards some that would directly benefit the LCA and the CADAC, in particular the Simulation Data Model (SimDB). This model contains a set of classes for describing simulations and simulation software, including descriptions of the goals and characterizing the results. When the IVOA does produce standards for theory data, this is model that will be used; therefore, this needs to be kept in mind as a design requirement, if we want to keep our tools interoperable in the future.

2. Design requirements

2.1. Service capabilities

To summarize the arguments of the previous section, the primary goal of any solution is to capture metadata about the simulation, as it's being produced. This metadata also needs to be rich enough to allow comparing simulations by different codes (the turbulence simulations),

[1] http://cadac.sdsc.edu

and simulations by a single code with different parameters (the galaxy cluster simulations). To be forward looking, we also want it to be interoperable with other VO services, and to have the potential to support an IVOA Standard if one becomes available. From the start, this leads us to think in terms of web services and XML, rather than, say, a custom desktop application.

Collecting all of these thoughts leads us to decide that the simulation catalog should:

- Be implementable as a web service.
- Capture sufficient detail to enable publishing to the VO as theory standards emerge.
- Be usable on a day-to-day basis for tracking simulations as they evolve.

Looking at these requirements together, a possible solution could be a web service providing access to a database. While that would be sufficient querying and maintaining the records in a database require thinking in terms of tables and rows, not simulations, parameters and datasets. I decided that an interface that provided a clearer representation of the underlying resources (or classes) would be preferred.

2.2. Client tools

To decide on a web service architecture, or style, it is necessary to consider who the users will be, what kind of environment (i.e., hardware, software) they will be operating in, and how much client software I wanted to write and maintain. To begin with, the users are expected to be researchers in computational astrophysics, most of whom have programming skill, some of it considerable. However, these skills are largely in numerical methods and high-performance computing, not web development, though almost all users have some experience building static web pages.

Like the skills of the researchers, the environment for running simulations is rather focused. Supercomputers and Linux clusters are almost universally shared systems with restricted users environments, occasionally with uncommon operating systems, and often administrated by an external organization, such as a supercomputer center. This makes installing additional software at least a tedious, if not a difficult or impossible task. This suggests that the client code will need to be lightweight, and easy to deploy on different platforms.

Finally, there is the amount of commitment I would able to provide to client code. My goal was to provide as little client up front as possible; I felt that maintaining both the service interface and client tools would lead to a coupling between the service the tools I provided, which would have a negative impact on future interoperability.

To match the skills and environment of the users, without building a suite of custom tools, the needs of the users seemed to restrict client-side tools to existing command-line applications such as cURL[2] or GNU Wget[3], or the URL and HTTP libraries that are now standard parts of both scripting languages such as Perl and Python. This would allow users to write their own client tools (perhaps with the help of some class libraries to handle data serialization) and incorporate them within their existing scripts for managing data.

3. A RESTful simulation catalog

Fortunately, RESTful web services meet the requirements outlined in Section 2.1. As stated earlier, REST is an acronym for *REpresentational State Transfer*, and was defined in a thesis by Fielding (2000). While not a standard, this definition encapsulates many of the practices used on the World Wide Web. For an in-depth introduction to REST, I recommend the book by Richardson & Ruby (2007), *RESTful Web Services*. It introduces not just the formal aspects of REST, but also an architectural style, called Resource Oriented Architecture (ROA) which is the one I adopted for the final design of this simulation catalog.

At its core, REST–and therefore ROA–is about manipulating (retrieving, creating, updating, etc.) *resources* (for example, the metadata about a simulation) through the *transfer* (e.g., downloading) of a *representation* (per-

[2] http://curl.haxx.se/

[3] http://www.gnu.org/software/wget/

haps a web page) of that resource. REST is agnostic as to the underlying system used to transfer and control the resources, while ROA restricts itself to the HyperText Transfer Protocol (HTTP), so as to be applicable to web services. At its simplest, downloading static web pages, ROA sounds trivial in today's world. The power comes from the fact that resources, whether web pages or something more abstract like a map, can be acted upon using the HTTP verbs, like GET, PUT, POST and DELETE, and that resources can be connected using URL (links).

Choosing ROA, and limiting the clients' interaction with the service to using HTTP to transferring representations of the simulation metadata means that our needs on the client side are met. As will be shown,

- RESTful services have simple, well defined interfaces, and
- are accessible using common software libraries (HTTP, XML, DOM, etc.).
- As a result, RESTful services are:
  - easy to access;
  - easy to duplicate.

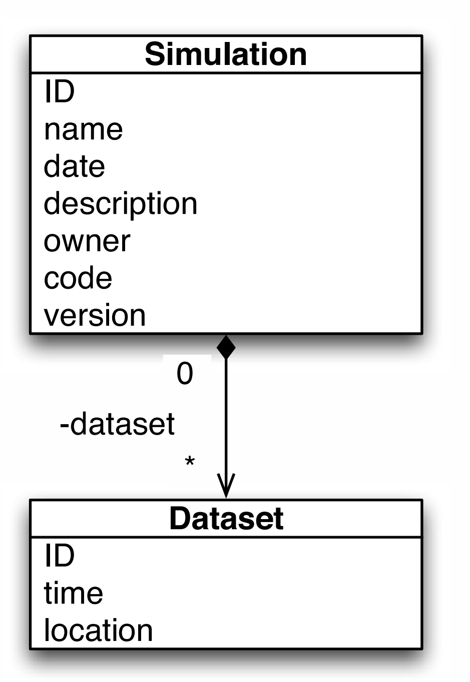

Fig. 2. UML diagram of the two resources in the example: simulations and datasets.

3.1. The service components

Because of the desire to provide a richer introduction to ROA, rather than describe SimCat in detail, I'm going to cover the fundamental concepts by designing a minimal web service to track simulations and datasets, basically a simplified version of the full catalog. Simultaneously, I'm going to introduce four components of ROA: resources; Universal Resource Locators (URLs); representations; and HTTP verbs.

3.1.1. Resources

Resources are the interesting things in the domain. For an online sky atlas, the resources may be maps of section of the sky, stars and constellations. In our example service, the resources are simulations and datasets. I tend to think of resources as being synonymous with classes, mainly for the help it provides during the design of the service. In particular, by diagramming the classes in Unified Modeling Language (UML), it's easy to understand the properties of resources, and their relationships.

Figure 2 shows a diagram of the classes for our example service. Each class has a few associated properties, such as an ID, the code that ran the simulation, and the location of the data. Because datasets are only produced by a single simulation, the simulations can be thought of as collections of datasets. (This is denoted in the diagram by the filled diamond at one end of the arrow.)

3.1.2. URLs

Uniform Resource Locators (URLs) serve two purposes in an ROA style web service: URLs are the names of the resources; and URLs provide links between resources. If you know the name (URL) of a resource, you can get a representation of it, just by pointing your browser at the address. It may be a less-than-useful representation–say, a binary file in a custom format–but at least you can access it. Similarly, if one resource references other resources, you can look at those, as well. This connection between resources is at the heart of the World Wide Web, and is one of the benefits of ROA.

Right now, we need to decide on the URLs for the simulations and their datasets. A clean and simple way is to begin with a base URL, e.g., `/simulations/`[4] , and append the ID of the simulation, which could be tied to a primary key in a database table, making `/simulations/1` refer to the first simulation in the catalog.

For the datasets, we could either repeat this, using `/datasets/` as our base URL, but because simulations collect datasets, I think `/simulations/1/datasets/` works better, and helps to show that a dataset belongs to a single simulation. This means that `/simulations/1/datasets/1` would refer to the first dataset of the first simulation.

One way to document the relationship of resources and URLs is URI templates; these provide a mapping from URIs to resources. At a minimum, it provides a means to document the service interface, which can then be implemented using pattern, often with regular expressions[5]. Simply put, URI templates place variable names inside of braces `{foo}`. This way, we can think in term of a generic simulation in our catalog, by writing the URL as `/simulations/{simulation}`, and treating `{simulation}` as the ID of the simulation. The same is true for datasets, i.e., `.../datasets/{dataset}` is the ID of a generic dataset.

Finally, if we consider list of simulations and datasets in our catalog to be resources (which we should), then URLs for those would be necessary. Two natural ones to use are `/simulations/`, to refer to a list of all the simulations in the catalog, and `/simulations/{simulation}/datasets/`, to name a list of the datasets for a single simulation.

[4] To keep the text cleaner, these URLs can all be presumed to be relative to a root URL, such as `http://catalog.example.edu`.

[5] In addition to documenting the interface, URI templates can be used to match requests, as is done by packages such as *selector* (http://lukearno.com/projects/selector/).

```
<?xml version="1.0" encoding="UTF-8"?>
<Simulation>
 <ID>1</ID>
 <name>GalCon2</name>
 <description>
   Milky Way-Andromeda collision
 </description>
 <owner>J. Dough</owner>
 <code>treecode</code>
 <version>1.4</version>
</Simulation>
```

Fig. 3. Sample XML file representing a simulation.

3.1.3. Representations

The data sent between the server and the client are representations of the resources. These could be in a variety of formats: XHTML, XML, JSON, CSV, etc. Which one to choose is tied to the intended application. For Ajax (Wikipedia 2008), in which the data is not intended to be human-readable, and the dominant programming language is JavaScript, JSON is the *de facto* standard. On human side of the web, where the resources are web pages to be rendered in browsers, it's HTML.

Following the design of the full simulation catalog, we'll choose another common format for our example catalog, XML. While we could have been very simple and used CSV, with each column referring to a different property of resource, XML tags provide a convenient self-description of the data. Figure 3 contains an example XML file for a fictitious simulation. A similar format could be used for the datasets, and the lists of resources.

An important point is that resources can have multiple representations. XML may be the appropriate one for a service that is intended to be used by a scripted client, but the service could also provide web pages about the simulations. The pages could be under a separate root URL, e.g., `/html/simualtions`, or

HTML pages could be return when the client uses the `Accepts` header in the request. This allows the resource to be available to separate parts of the Web: the programmable one, which we associate with web services; and the human one, driven by web browsers.

3.1.4. HTTP verbs

Resources identified by URIs are accessed and modified using the standard HTTP methods. In a sense, the HTTP verbs define interface to the service, these verbs determine what will happen when a particular URL is requested. With one exception (which will be covered shortly), all ROA style service should behave uniformly, depending on which verb is part of the request. This uniform interface is part of what makes RESTful services simpler than SOA ones: they all behave similarly. (In principle, once you know the resources and URLs of a service, you totally understand the service. Everything else is covered by HTTP.)

For SimCat (and our example service), four of the verbs will be the most important:

`GET` Retrieve a representation of the resource.
`PUT` Create a new resource.
`POST` Modify an existing resource, or append to a list of resources.
`DELETE` Delete a resource.

The verbs `GET` and `DELETE` should be easy to understand. `GET` is the verb sent by web browsers in their requests to retrieve web pages. In our example, a `GET` request to `/simulations/1` would return the XML document representation the simulation with the ID of 1[6]. Likewise, sending a `DELETE` request to `/simulations/1` would delete that resource. A `PUT` request to `/simulations/1`, along with an XML describing the simulation, would create a new simulation with an ID of 1 (unless a simulation with that ID already exists, in which case the server would either overwrite the previous one, or respond with HTTP error code 409). `POST` can be used in two ways: to update, or modify a resource; or to add a new resource to a list, perhaps the datasets of a simulation. When updating a resource, an XML fragment can be sent containing only the properties to be updated.

[6] If there is no simulation with an ID of 1, the service should return an HTTP error code of 404. This is another convenient part of choosing use HTTP as more than just a transportation layer for our data: Many of the error codes are already described for us.

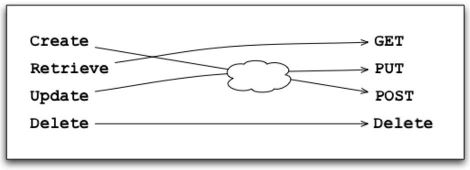

Fig. 4. Comparison of the common verbs used in databases (create, retrieve, update, delete), and the ones used on the web (GET, PUT, POST, DELETE).

Not all of the resource need to respond to every verb. For example, the lists of simulations may only accept the `GET` verb, requiring that each individual resource be deleted in order to remove all of them.

There is an analogy between RESTful services and databases, particularly for something like a catalog. The individual resources are similar to the rows in a database table. People working with databases sometimes use the acronym CRUD (Create, Retrieve, Update, Delete), to summarize the possible actions when dealing with rows of a database. Figure 4 shows the connections between these actions and the four HTTP verbs.

The cloud in Figure 4 is the source of non-uniformity in RESTful service behavior mentioned earlier. There is some disagreement amongst developers about how `PUT` and `POST` should be used for creating and updating resources. The resolution is to document how services you create use these verbs, and to read the documentation of services you need to access.

4. SimCat

The extension of the basic service we designed above is SimCat[7]. SimCat is the software implementing of a service with resources for tracking and cataloging simulations and simulation data. Because the primary purpose of this paper was to introduce REST and ROA as good tools for building catalogs in the VO, here we will only highlight the major details of SimCat.

4.1. Resources

The most significant differences between SimCat and the example service are the number of resources, and that the SimCat resources are modeled as classes that inherit from those in the SimDB. Here are the major resources (or classes) in SimCat:

- programs;
- computers;
- runs;
- datasets;
- simulations;
- and projects.

Figure 5 shows an overview of the relationship of these resources. It is important to note that this summary of the SimCat resources leaves out many of the ones used for characterization of the simulation data, as found in the SimDB. The reason for this is simply brevity.

The additional resources beyond the SimDB are the runs and computers. During the day-to-day process of running a simulation, users are mostly concerned with where the simulation is running, and how far it has progressed. These resources allow them to maintain the status of their simulations by treating them as a series of runs on various computers. Once the simulation is complete, this information is not necessary, which is why the classes are not included in the SimDB model, which is only concerned with describing existing data.

[7] http://code.google.com/p/simcat

4.2. URLs, representations, verbs

The details of the SimCat interface can be found on the project web site[7]. The URLs are described in terms of URL templates on the wiki pages, for example computers are named by `/computers/{computer}/`. The resources are serialized to and from XML instance documents. Each of the resources has an associated XML Schema, which defines the elements of the documents, and can be used to validate them. Finally, the service handles the HTTP verbs in the same way as the example service.

4.3. Implementation

A prototype was developed using a combination of *selector* and the Python `wsgiref`, and run behind the Apache web server using `mod_rewrite`. An XML style sheet was used to present the resource in XHTML, in addition to the XML format. This way, web pages about the simulations could be incorporated into our current web site.

Presently, SimCat is being written on top of the Django web framework[8]. Django projects are collections of separate applications (or apps). This is very convenient, since it has allowed me to model the SimDB classes in a separate application, and then inherit them into the SimCat application. As other theory-specific standards come out of the VO, I expect this pattern to be repeated.

5. Summary

Hopefully this paper has met its purposes, to present REST as an appropriate style for web services in the VO, and to introduce SimCat, a RESTful Simulation Catalog. This work is a continuation of our efforts to publish the data produced by the LCA to the Virtual Observatory (Wagner & Norman 2006). One result of my experience in building web services using both SOA and ROA, is a willingness to recommend ROA as a style appropriate for small research to publish data (particularly online catalogs). REST and ROA are easy

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

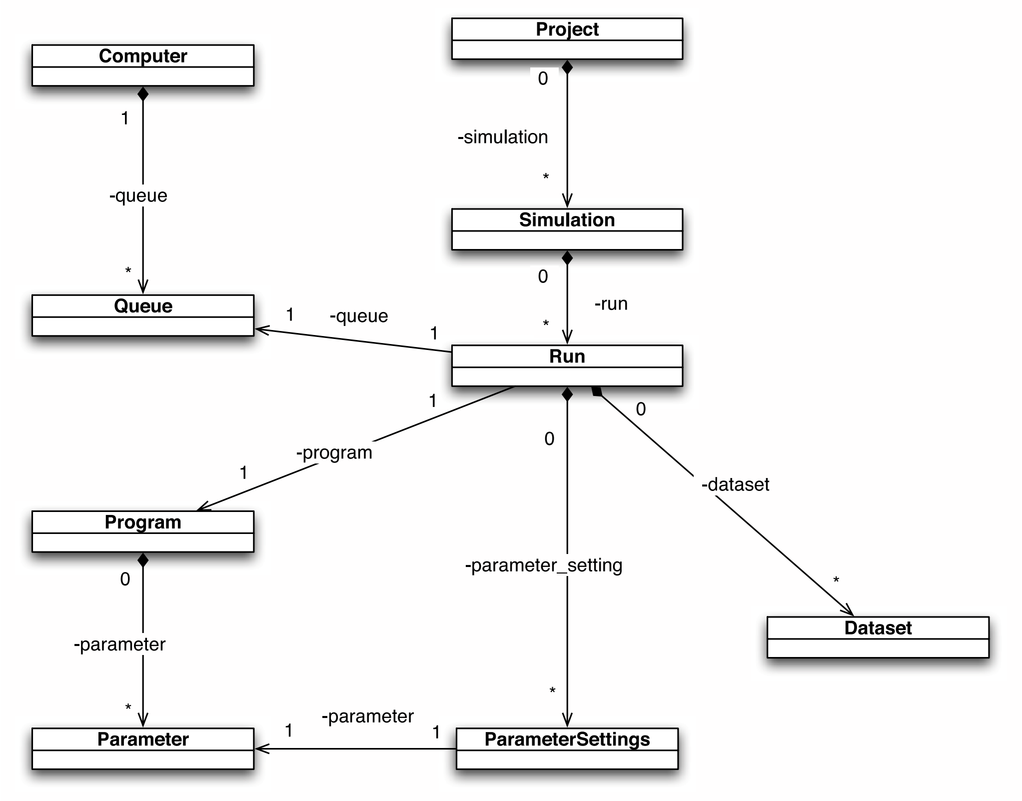

Fig. 5. Summary of major SimCat resources.

to understand, and reuse the same patterns that we are familiar with on the World Wide Web. RESTful services are easier for users with minimal web development experience to access, and client software can be built using libraries included with several open source programming languages.

To help ensure that our tools will be interoperable with others, SimCat extends the classes in the SimDB, and adds resources for tracking simulations as they progress, making it useful on a daily basis. My goals in all of this are to build on the work being done in the IVOA, and to provide tools for the CADAC, and other research groups involved in computational astrophysics.

Acknowledgements. I wish to thank the organizers of the workshop for inviting me, and EuroVO for providing the funding for my travel.

References

Fielding, R. T. 2000, PhD thesis, University of California, Irvine

Richardson, L. & Ruby, S. 2007, RESTful Web Services (O'Reilly and Associates)

Wagner, R. P. & Norman, M. L. 2006, in Bulletin of the American Astronomical Society, Vol. 38, Bulletin of the American Astronomical Society, 1002

Wikipedia. 2008, Ajax — Wikipedia, The Free Encyclopedia, [Online; accessed 11-January-2008]