



Cloud Application Services Platform (CLASP): User Guide, Introduction, and Operation

Craig Sayers, Hernan Laffitte, Prakash Reddy, Kivanc Ozonat, Mehmet Sayal, Alkis Simitsis, Sharad Singhal, Georgia Koutrika, Mahashweta Das, Ablimit Aji, Hitesh Amrutlal Bosamiya, Marcelo Riss, Kevin Wilkinson, Jose Clemilson de O. Lucio, Alyne Gomes Soares Cantal, Carlos Rosemberg Maia de Carvalho

HP Laboratories

HPL-2015-76

Keyword(s):

Cloud Catalog; Service Catalog; App Catalog; Cloud services; SaaS; PaaS; R; Python

Abstract:

Software developers at large tech companies spend a lot of time writing code for tasks that colleagues elsewhere in the organization have already addressed. Scripts are rarely written or documented with discovery in mind, and the APIs on which they depend are frequently inconsistent, further limiting reuse. For mobile devices the App Catalog serves as an essential intermediary, streamlining the process both for developers and end users. We've created an experimental platform called CLASP (Cloud Application Services Platform) applying that model by publishing services and datasets instead of apps. It includes support for existing APIs, and we've also created an SDK (software development kit), so our users can write other operations themselves and easily publish in the catalog for later discovery and reuse. CLASP allows us to take a very diverse set of operations and make them all available through a consistent compositional interface. For example, you can retrieve log messages using OneView and analyze the text using Autonomy, or gather system configuration using iLO interfaces and persist the results in a Vertica Database. Our internal deployment now has more than 2,000 services, and has been used by more than 150 developers. It allows application developers to discover, test, and use services while providing a seamless app-catalog-type experience for service developers, allowing them to code and test locally while semi-automating the process of publishing those in the catalog. Developed by HP Labs, CLASP is an experimental platform and not a product. This report includes an introduction, quick-start guide, and implementational details.

External Posting Date: September 6, 2015 [Fulltext] Approved for External Publication

Internal Posting Date: September 6, 2015 [Fulltext]

CLASP

Cloud Application Services Platform

USER GUIDE, INTRODUCTION, AND OPERATION

Craig Sayers, Hernan Laffitte, Prakash Reddy, Kivanc Ozonat, Mehmet Sayal, Alkis Simitsis, Sharad Singhal, Georgia Koutrika, Mahashweta Das, Ablimit Aji, Hitesh Amrutlal Bosamiya, Marcelo Riss, Kevin Wilkinson

HP

José Clemilson de O. Lúcio, Alyne Gomes Soares Cantal, Carlos Rosemberg Maia de Carvalho

Atlantico

Contact: Craig.Sayers@hp.com or Sharad.Singhal@hp.com

Summary

Software developers at large tech companies spend a lot of time writing code for tasks that colleagues elsewhere in the organization have already addressed. Scripts are rarely written or documented with discovery in mind, and the APIs on which they depend are frequently inconsistent, further limiting reuse.

For mobile devices the App Catalog serves as an essential intermediary, streamlining the process both for developers and end users. We've created an experimental platform called CLASP (Cloud Application Services Platform) applying that model by publishing services and datasets instead of apps. It includes support for existing APIs, and we've also created an SDK (software development kit), so our users can write other operations themselves and easily publish in the catalog for later discovery and reuse.

CLASP allows us to take a very diverse set of operations and make them all available through a consistent compositional interface. For example, you can retrieve log messages using OneView and analyze the text using Autonomy, or gather system configuration using iLO interfaces and persist the results in a Vertica Database.

Our internal deployment now has more than 2,000 services, and has been used by more than 150 developers. It allows application developers to discover, test, and use services while providing a seamless app-catalog-type experience for service developers, allowing them to code and test locally while semi-automating the process of publishing those in the catalog.

Developed by HP Labs, CLASP is an experimental platform and not a product.

This report includes an introduction, quick-start guide, and implementational details.

Contents

- Summary 2
- Introduction..... 5
 - Service developers 5
 - Application developers 5
 - Datasets 5
 - Existing APIs 5
 - Deployment..... 5
 - Overview..... 5
- Background..... 6
- Terminology..... 9
- Introductory Demonstration and Quickstart Guide 10
 - Application Embellishment using Services 10
 - Customization by Service Composition 14
 - Service Development Using R, Python, JavaScript, or Java..... 24
 - Dataset Ingestion..... 27
- Accessing External APIs..... 33
 - Custom services for external APIs 33
 - Bulk Ingestion of services for external APIs 34
- Implementation and Infrastructure..... 35
 - Logical View..... 35
 - Physical View 35
 - Security..... 36
 - Speed 37
- Service Definition 39
 - Common parameter definitions 40
 - Elementary service parameters..... 42
 - Java Hello World Example 43
 - Scripted Fast Fourier Transform Example 44
 - Composition Definition and Example..... 45
- Multiple Catalogs and Sharing Services..... 48
- Internal Service Execution Process..... 49
 - Elementary services..... 49
 - Composed services..... 49

Semi-automated descriptions and schemas	51
Example script	51
Schema extraction and cataloging.....	51
Generating human-readable descriptions	53
Optimizing composition using schemas	55
Automated text and compositions	55
User experience	56
Conclusions.....	59
Acknowledgements.....	60

Introduction

The Cloud Applications Services Platform (CLASP) is a catalog of services and datasets with back-end support for execution and composition. We take the easy development process and convenient discovery from mobile app catalogs and use it for the back-end RESTful-style services on which many applications depend. The resulting catalog aids application development, simplifies the publication of algorithms as cloud services, and supports composition with datasets and external RESTful services. Developed by HP Labs, it is an experimental platform and not a product.

Service developers

Writing a great new algorithm is hard. It is a regrettably uncommon skill. And so we'd prefer algorithmic experts focus on that task without undue distraction while also allowing the results of their labor to be shared with the community. Using our catalog, algorithm developers can experiment using familiar tools (for example Python, or the R workbench), test using local files, and then publish the result as a web service in just a few minutes without any web programming skills.

Application developers

Writing great applications requires very different skills but is no less difficult. We'd similarly prefer those experts focus on application development without needing to learn the internal workings of every necessary algorithm. Using our catalog, application developers can discover suitable services, test them via the website on their own data to verify efficacy, and then just copy/paste our generated example code to add the service to their applications.

Datasets

Both algorithms and applications need data, and so our catalog also manages that. We support two basic styles of data: Descriptive, and Managed. Descriptive Datasets serve as search aids in the catalog, aiding users in discovering services by describing the hidden data on which they depend, while Managed Datasets are stored on our catalog database servers and available for direct SQL-based access.

Existing APIs

We also recognize the breadth of existing RESTful style HTTP-based interfaces and so we also support adding those to our catalog. Accessed via thin wrappers, they can be discovered, composed, and used by application developers in exactly the same way as native platform services.

Deployment

Our internal deployment follows a conventional web server model with a load balancer covering a number of back-end processing servers, with shared databases behind that. The system now has more than 2,000 services, and has been used by more than 150 developers.

Overview

This report includes some background, a quick-start guide, and implementational details.

Background

This section provides some background on the motivation for our Cloud Application Services Platform (CLASP) driven by experiences with book processing and mobile application development. Readers more interested in using the system may safely skip this section upon first encounter.

Several years ago, there was a project called BookPrep which took scans of over a million old books and, with sophisticated image processing algorithms, prepared the raw scans for use in a print-on-demand solution. Examples of the required image processing are shown in Figure 1.

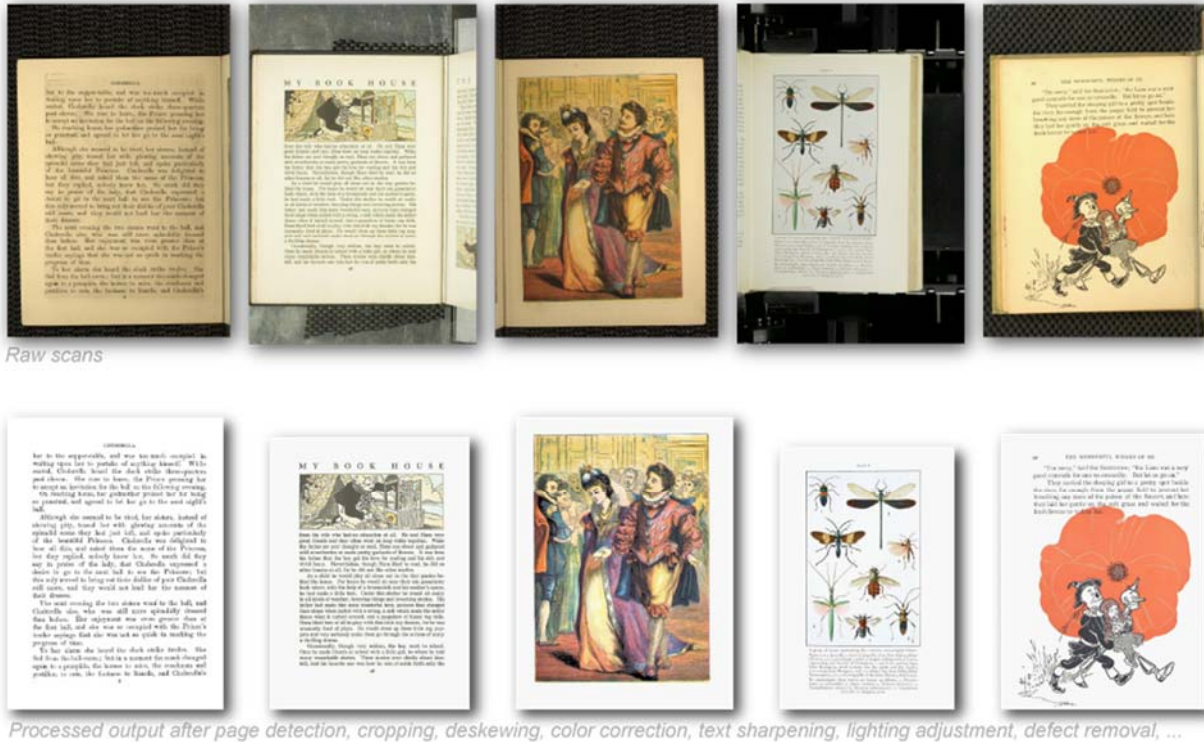


Figure 1. Input and output images for BookPrep processing, taking raw image scans (top row) and preparing them as printable content (bottom row).

In a not uncommon experience, developers observed the difficulty in finding and reusing image processing components developed by their peers. Even in the simple case of image scaling, the algorithms are surprisingly complicated. Writing an image scaler is easy, but writing one which can generate the best possible output under particular constraints is a non-trivial endeavor. Having that sophisticated algorithm developed once, by a team of experts is preferable.

In addition, they needed to be concerned not only with the image processing algorithms, but also with the complexities of demand estimation, cloud deployments, processing, and security.

To address those issues, we desired a system which would simplify the discovery and reuse of previous development investment, while also providing an execution environment with automated cloud deployment.

In parallel with the BookPrep back-end processing, we were becoming familiar with the WebOS Application Catalog (see Figure 2), both as users (developing the BookPrep application for the Touchpad) and as developers (experimenting with recommendations to aid catalog users). In that process we came to appreciate how desirable the Mobile Application Development model is.

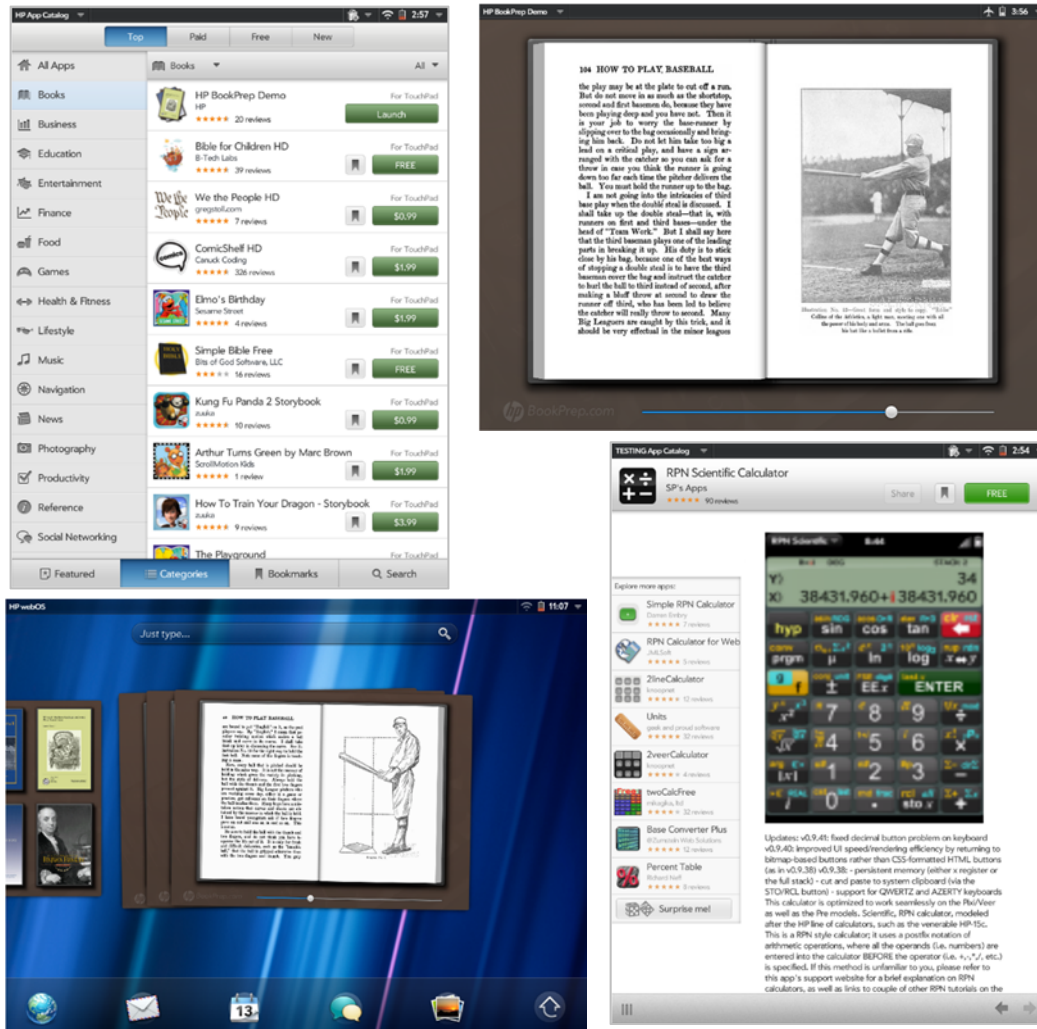


Figure 2. Example of the Bookprep application for WebOS (top right) which benefited from application discovery in the catalog (top left) and from the WebOS framework giving the ability to read multiple books (bottom left), and experimental recommendations in the WebOS app catalog (bottom right).

Writing mobile apps is simplified by an SDK providing a basic set of components common to many applications. Conveniently, developers need only test locally on a single device and do not need to predict demand in advance. Applications further benefit from the execution platform itself. For example, in the case of WebOS, the cards-based interface allowed users to use our BookPrep application to have multiple books open simultaneously without any explicit support for that within our code.

Along with distribution, App Catalogs aid in discovery, both through direct access (when users already know the application they seek) and indirectly (where the catalog supports serendipitous discovery by aiding users in faceted browsing, advanced search, and recommendations).

Our goal was to take the easy development process and convenient discovery from the mobile catalog and use it for the back-end RESTful-style services such as those needed for the BookPrep image processing.

For application developers we provide:

- Discovery of services via search, categorization, and recommendation
- A simplified programming model with consistent RESTful APIs across a range of domains
- The ability for prospective users to test services with their own data
- Example code to invoke the services suitable for cut-and-pasting directly into applications
- The ability for simple customization via service composition without coding

For service developers we provide:

- An easy drag/drop wizard interface for adding new services to the catalog
- Support for Python, R, and Java
- The ability to test locally on a single machine
- Common building blocks including the ability to process input parameters, perform i/o from/to a variety of sources, while supporting logging and error handling.
- A runtime execution environment in the cloud with automated deployment
- The automatic generation of example code and test interfaces
- The automatic ability for any service to be used in compositions

Over time we expect a virtuous cycle where the addition of services encourages catalog use by application developers, that larger audience will then in turn encourage more service developers to publish their work.

Our CLASP catalog now has more than 2,000 services. It allows application developers to discover, test, and use those, while providing a seamless app-catalog-type experience for service developers, allowing them to code and test locally while semi-automating the process of publishing those in the catalog.

Terminology

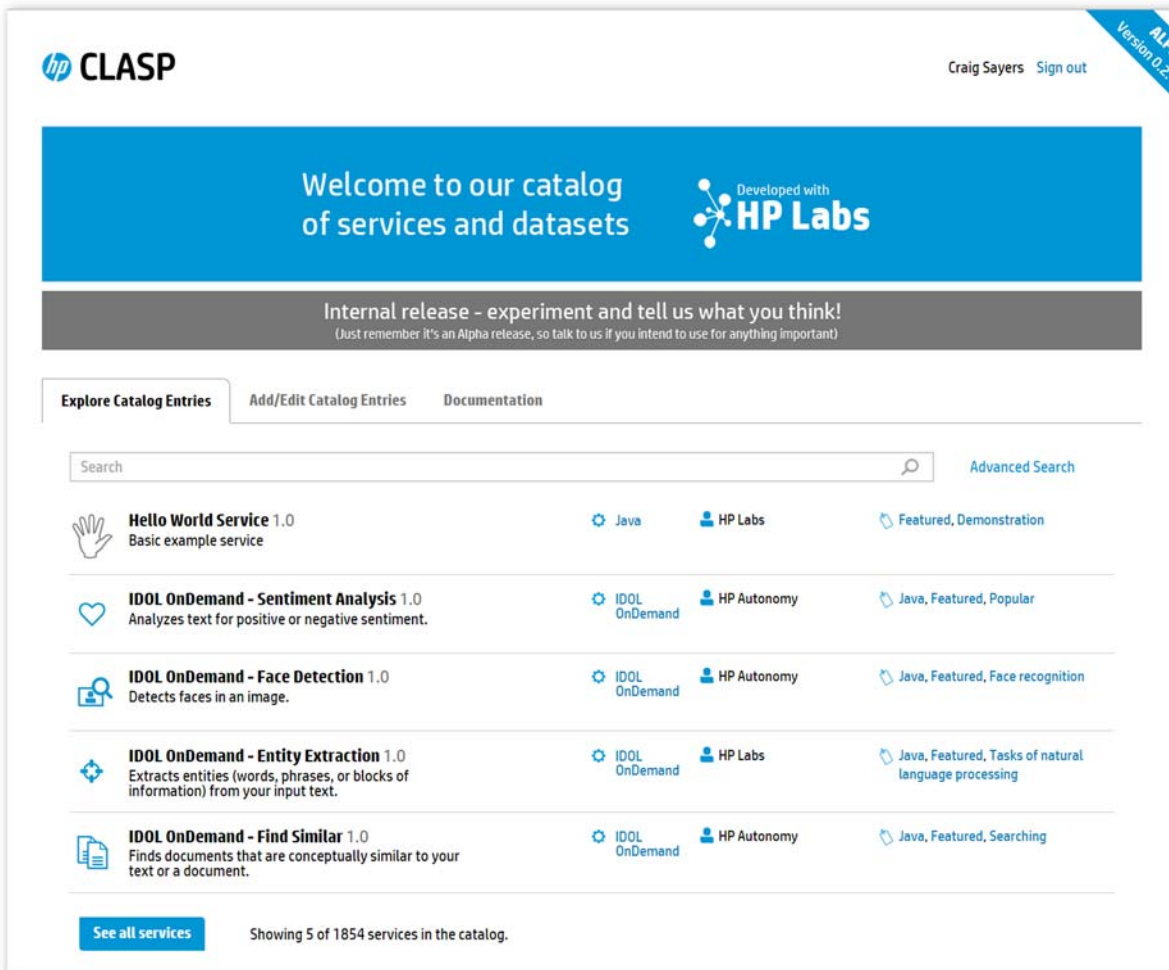
- Service** Each service in our system is small, typically performing a single operation. For example an image scaling service would take an input image and scale it to produce an output image.
- Task** A single execution of a service on specified input parameters to produce specified output. For example, an image scaling task would take a specific input image and scale it, storing the output in a particular specified location.
- Composition** The special case of a Service constructed from one or more other services. Typically made via the compositional user interface, these provide customized service creation without the need for any programming. They are easier to construct, but less powerful than custom coded services. Each Composition is itself a Service, and can thus be used and composed just as any other catalog Service.
- Dataset** An entry in the CLASP Catalog describing data rather than a Service. This is typically tabular data, containing a number of rows and columns where all the data in each column is of the same type, as one might find in a single Database table.
- R** A popular math programming language, having basic types including Vectors and Matrices, and a large number of optimized libraries for common mathematical algorithms. It is not uncommon for university math and science graduates to have used this in their coursework.
- Python** A popular general-purpose interpreted programming language offering a good selection of powerful features derived from other languages and supporting easy extensibility. It has many applications, but in our case is particularly well-suited for execution on back-end servers.
- JavaScript** A programming language commonly used to code operations inside web pages for execution within the client web browser. (Despite the name and superficially-similar syntax, it is quite different than Java, offering more power and flexibility but much less structure.) It is best-suited to small flexible programs.
- JSON** The JavaScript Object Notation, a data format for passing data using a familiar JavaScript syntax.

Introductory Demonstration and Quickstart Guide

The section presents several illustrative examples of catalog usage from the perspective of application and service developers. Using actual screenshots from the working system, it provides both an initial introduction and serves as a quick-start guide. If you wish to use CLASP, this is a quick tutorial to gain familiarity, while if you're investigating CLASP it's a simple introduction.

Application Embellishment using Services

To begin embellishing your application, use the search mechanism in the catalog to look for services of interest. You can search for text words and/or click on any category tag to limit the results:



The screenshot shows the CLASP catalog interface. At the top left is the 'hp CLASP' logo. At the top right, it says 'Craig Sayers Sign out' and 'Version 0.2.1' in a blue banner. Below the logo is a blue banner with the text 'Welcome to our catalog of services and datasets' and 'Developed with HP Labs'. Below this is a grey banner with the text 'Internal release - experiment and tell us what you think! (Just remember it's an Alpha release, so talk to us if you intend to use for anything important)'. Below the banners are three tabs: 'Explore Catalog Entries', 'Add/Edit Catalog Entries', and 'Documentation'. Below the tabs is a search bar with the text 'Search' and a magnifying glass icon, and a link to 'Advanced Search'. Below the search bar is a list of five services, each with an icon, a title, a description, and tags. The services are: 'Hello World Service 1.0' (Basic example service, Java, HP Labs, Featured, Demonstration), 'IDOL OnDemand - Sentiment Analysis 1.0' (Analyzes text for positive or negative sentiment, IDOL OnDemand, HP Autonomy, Java, Featured, Popular), 'IDOL OnDemand - Face Detection 1.0' (Detects faces in an image, IDOL OnDemand, HP Autonomy, Java, Featured, Face recognition), 'IDOL OnDemand - Entity Extraction 1.0' (Extracts entities (words, phrases, or blocks of information) from your input text, IDOL OnDemand, HP Labs, Java, Featured, Tasks of natural language processing), and 'IDOL OnDemand - Find Similar 1.0' (Finds documents that are conceptually similar to your text or a document, IDOL OnDemand, HP Autonomy, Java, Featured, Searching). At the bottom left is a blue button 'See all services' and at the bottom right is the text 'Showing 5 of 1854 services in the catalog.'

Then just click on any service title to see a summary.

In this case we'll pick the simplest service in the catalog:

The screenshot shows the 'Hello World Service 1.0' page in the 'Summary' view. The page title is 'Hello World Service 1.0' with the subtitle 'Basic example service'. A hand icon is in the top left. On the right, there is a 'Share' button, 'Developer: HP Labs', 'Tags: Demonstration, Featured, Java', and 'Related Services: IDOL OnDemand - Sentiment Analysis, IDOL OnDemand - Face Detection, IDOL OnDemand - Entity Extraction'. The main content area shows a service diagram with an 'input string' and an 'output message string'. Below the diagram, it says 'This is a very basic example service which simply says Hello.' and a yellow warning box states 'This service makes no security assertions, use with caution.' On the left, a navigation menu has 'Summary' selected, with other options: 'Details', 'Try', 'Call', and 'Compose'.

and use the tabs on the left to view details:

The screenshot shows the 'Hello World Service 1.0' page in the 'Details' view. The layout is similar to the summary view, but the 'Details' tab is selected in the left navigation menu. The service diagram is shown in more detail. Below the diagram, the input and output are defined: 'input string any text name' and 'output message string The output salutation'. The text 'This is a very basic example service which simply says Hello.' is now positioned to the right of the diagram. The yellow warning box remains at the bottom. The right-hand side information (Share, Developer, Tags, Related Services) is identical to the summary view.

test the service using your own input:

Explore / Hello World Service 1.0

Hello World Service 1.0

Basic example service

Summary
Details
Try
Call
Compose

You can try this service now using our data if you wish. If you're new to the system, then try the default wizard first, it will guide you through the process.

Default Wizard
Walks you through the process of testing the service.

1 Provide inputs values > 2 View result

input *

any text name

Submit

Share

Developer
HP Labs

Tags
Demonstration
Featured
Java

Related Services
IDOL OnDemand - Sentiment Analysis
IDOL OnDemand - Face Detection
IDOL OnDemand - Entity Extraction

examine the result:

Explore / Hello World Service 1.0

Hello World Service 1.0

Basic example service

Summary
Details
Try
Call
Compose

You can try this service now using our data if you wish. If you're new to the system, then try the default wizard first, it will guide you through the process.

Default Wizard
Walks you through the process of testing the service.

1 Provide inputs values > 2 View result

Task Id: 9ab3a154-b408-4d91-910b-b25b641de988
Your task has been processed.

output message

Do a new test

Share

Developer
HP Labs

Tags
Demonstration
Featured
Java

Related Services
IDOL OnDemand - Sentiment Analysis
IDOL OnDemand - Face Detection
IDOL OnDemand - Entity Extraction

and find example code for invoking the service which you can cut/paste into your app:

The screenshot shows the 'Hello World Service 1.0' page in the Oracle Cloud Developer Catalog. The page title is 'Hello World Service 1.0' with the subtitle 'Basic example service'. The main content area is titled 'Here you can find some example code for calling this service from Java and JavaScript applications.' and is divided into two sections: 'JavaScript Code' and 'Java Code'. The 'JavaScript Code' section includes a jQuery AJAX call example. The 'Java Code' section includes a simple string construction example. The page also features a sidebar with navigation options (Summary, Details, Try, Call, Compose) and a right-hand sidebar with developer information (HP Labs), tags (Demonstration, Featured, Java), and related services (IDOL OnDemand - Sentiment Analysis, IDOL OnDemand - Face Detection, IDOL OnDemand - Entity Extraction).

```
1 // Construct the task header (which depends on the service being called)
2 var taskHeader = {
3   "mode": "synchronous",
4   "services": [{"id": "com.hp.clasp.example.helloWorld", "version": "1.0", "engine": "Java"}]
5   //, "timeout": 2.5 // This is an optional longer timeout in seconds in case your service will take a long time
6 };
7
8
9 // Construct the task data containing your particular input and output values
10 var taskData = {
11   // inputs and outputs here, e.g.:
12   "inputName": "CLASP User"
13 };
14
15 // Build the URL
16 var taskURL = "https://clasp.hpl.hp.com/cfe/queue/1.0?" +
17   "taskHeader=" + encodeURIComponent(JSON.stringify(taskHeader)) + "&" +
18   "taskData=" + encodeURIComponent(JSON.stringify(taskData));
19
20 // And then retrieve it
21 // This used the jQuery ajax call (you may need a POST if the task data is large)
22 $.ajax({
23   type: "GET",
24   url: taskURL,
25   contentType: "application/json",
26   dataType: "json",
27   cache: false,
28   error: function() { alert("Something failed!"); },
29   success: function(jsonStatus) {
30     alert("success!" + JSON.stringify(jsonStatus));
31   }
32 });
33
```

```
1 // Construct the task header (which depends on the service being called)
2 String taskHeader = "(" +
3   "\mode\": \"synchronous\",\" +
4   \"services\": [{\"id\": \"com.hp.clasp.example.helloWorld\", \"version\": \"1.0\", \"engine\": \"Java\"}]\" +
5
```

In the preceding example, the output parameter was a simple string, so no user input was required, but more generally, the output would be a stream defined by a mime type, for example **text/csv** for a comma-separated-values file, or **application/json** for a JSON-formatted text file. In those cases, the system needs to know where to store the service output. For that we use URIs. These will vary based on the CLASP installation, but the one common to all installations has a CLASP-specific URI scheme of “task”, which asks CLASP to create a task-specific storage location, place the output there, and then return a URL to that location in the results. Those task-specific locations are temporary and intended to hold the results for a few hours until your application is ready to accept it. For example, if the output is of type ‘application/json’, and the requested output was ‘task:out.json’, then after the service runs, the result will replace that URI with a URL of the form:

https://clasp. com / ... someLongRandomString ... /out.json

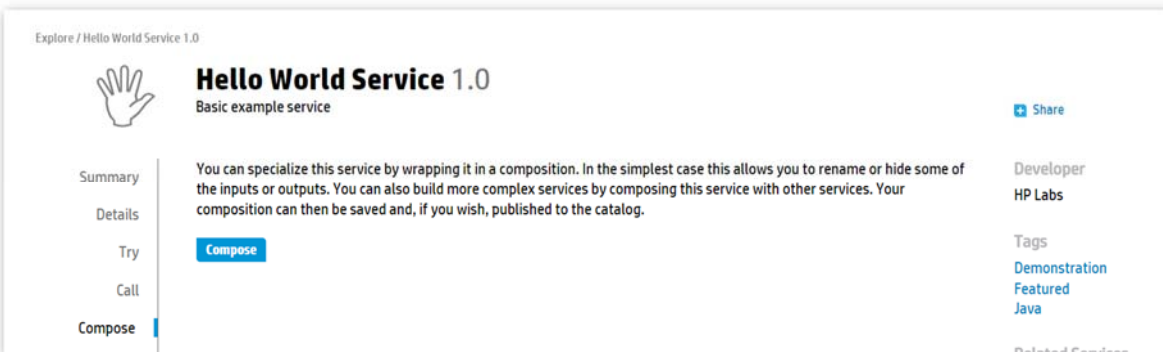
If no single service exactly meets your needs, it may nevertheless be possible to achieve a satisfactory result by combining together several catalog services. This process is called Composition, and is accessible on any service in the catalog via the Compose tab.

Customization by Service Composition

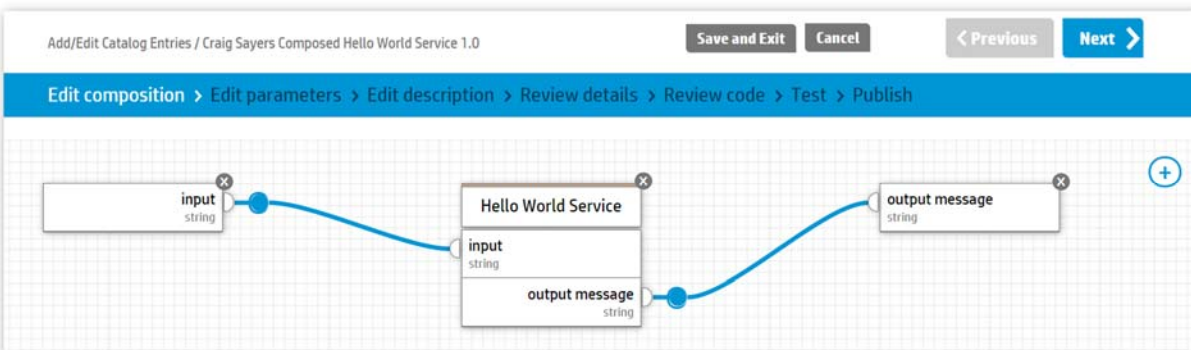
The composition of existing services is a simple but powerful technique for creating customized operations, permitting construction without writing any code or understanding the internal working of each service.

Here we combine Hello World with String Concatenation; using trivial services to focus on the creation and publication process. Real-world compositions are typically more complex, but follow the same process.

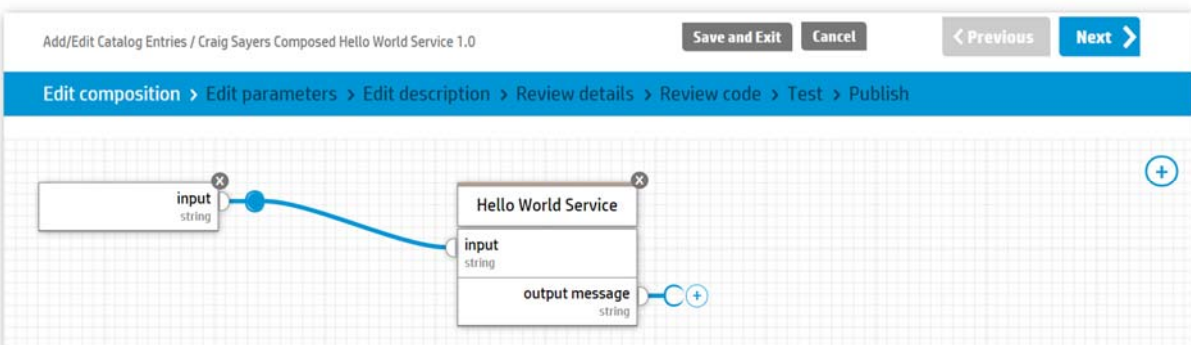
The first step is to start with any existing service:



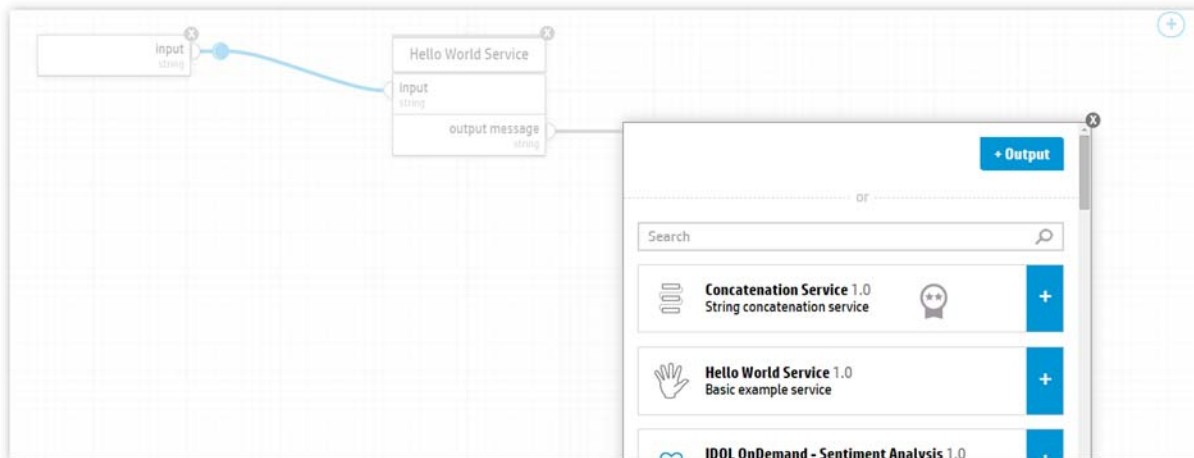
After clicking the Compose button, the system will create an initial composition with that source service as a starting point and its inputs and outputs will be used to create parameters for the composition.



Additional inputs, outputs, or component services can be added, while any unnecessary parameters or services can be deleted. Here we'll delete the output message:

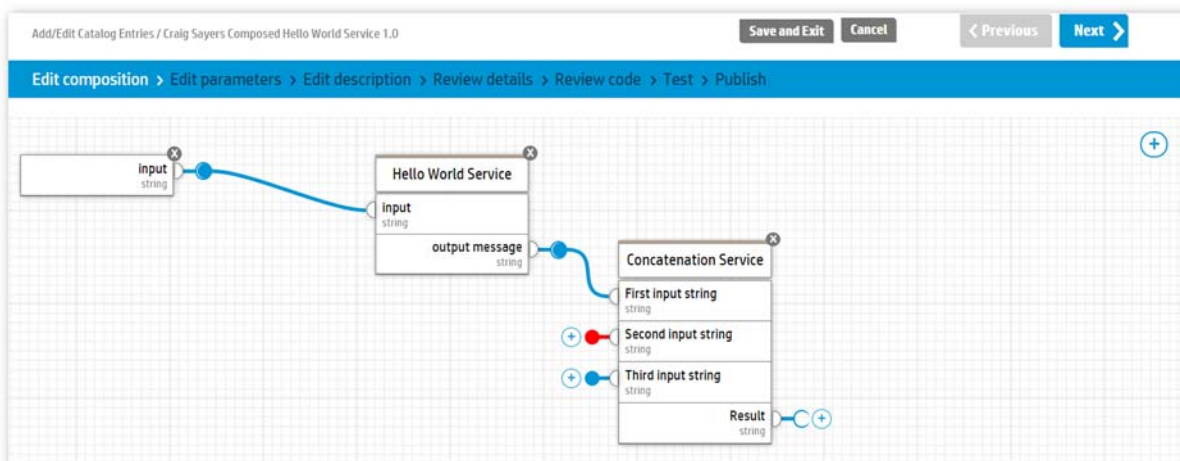


and click on the resulting '+' to see a list of relevant services:



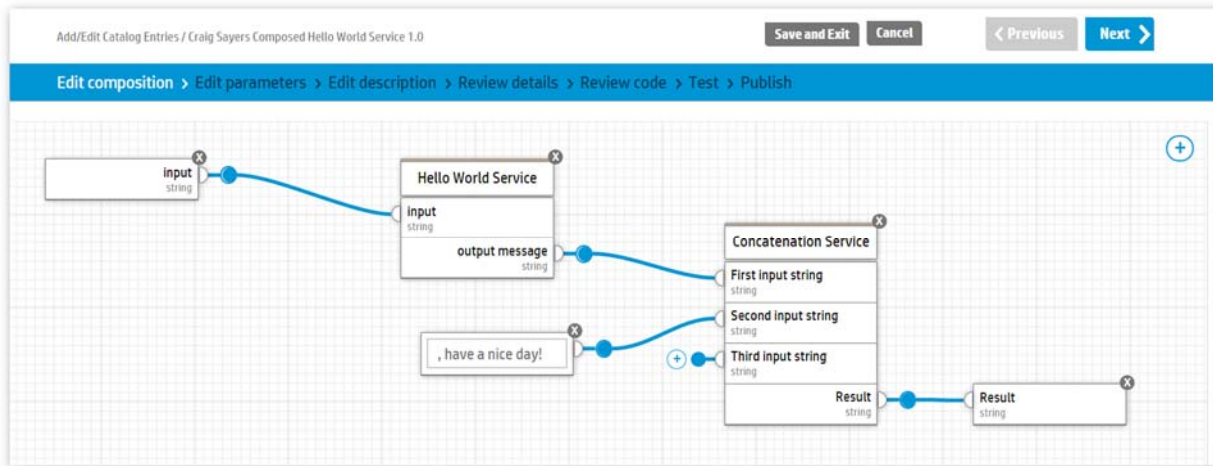
That list is restricted to those which are type-compatible, and then sorted based on past catalog usage by all users. Service combinations which are particularly common are visually indicated, for example in this case the concatenation service in the list has a badge indicating frequent past catalog usage by other users. Initially there is an unfortunate dearth of past usage. To partially overcome this we can mine external sources, extracting initial recommendations. For example, in the case of services which correspond to R functions, we mine externally-visible R code, using frequency of co-occurrence and code proximity as an initial estimate of the usefulness of connectivity in compositions. This is of course only an estimate and given an appropriately low weighting so it initially influences the list but is later swamped by actual catalog usage.

Returning to our example, the concatenation service is selected and added to the composition:



Any errors, for example a missing required input, are marked in red. Connections can be adjusted by dragging endpoints and adding additional inputs, outputs, or services. In this particular case the concatenation service requires at least two input strings, but only one is connected, hence the need for adjustment. Notice also that there is no output. That is not marked as an error, since it's not unreasonable to have services (such as writing to a database) which perform a side-effect without producing any output parameter. There is no need for additional outputs to indicate success or failure since those are handled via an exception mechanism. If any component service fails the entire composition is terminated with an appropriate error explanation being returned to the caller.

In this case we'll complete the composition by adding a string constant for concatenation, and an output for the result.



Once satisfied, click 'Next' to advance through the process of publishing your service in the catalog beginning with editing parameter details.

This screenshot shows the 'Edit parameters' tab of the service catalog. The breadcrumb trail is 'Edit composition > Edit parameters > Edit description > Review details > Review code > Test > Publish'. A message at the top reads: 'To prepare for publishing, make any necessary edits in the parameter descriptions and then click Next to proceed.' The 'Inputs' section lists 'input (inputName) string' and 'Second input string (inputString2) string', with an 'Add input' button. The 'Operation' section states: 'This service is a composition (use the Edit composition tab to see inside).' The 'Outputs' section is expanded to show the 'Greeting (outputString) string' parameter. It includes fields for 'Parameter identifier' (outputString), 'Parameter name' (Greeting), and a 'Required parameter' checkbox. The 'Parameter type' is set to 'String'. The 'Parameter description' field contains 'An uplifting salutation'. There is also a 'Default value' field. The interface includes buttons for 'Save and Exit', 'Undo', '< Previous', and 'Next >'.

The parameter information is pre-populated with information derived from component services and you will frequently need to customize that. For example in the output we've changed the description from the general string concatenation to one more appropriate for this composition and we may similarly adjust the input


name and description. A common technique is taking general services, wrapping in compositions, and then customizing the human-readable names and descriptions to produce domain-dependent services.

Once satisfied with the human-readable parameter descriptions, it's time to edit the catalog description, adding an icon as necessary, and editing the human-readable text to describe the entire operation for your composed service.

Add/Edit Catalog Entries / Polite Greeting Service 1.0 Save and Exit Undo < Previous Next >

[Edit composition](#) > [Edit parameters](#) > [Edit description](#) > [Review details](#) > [Review code](#) > [Test](#) > [Publish](#)

To prepare for publishing, make any necessary Edits, then Review and Test the catalog entry.

 **icon.png**
Icons must be a PNG file and we recommend 256x256 pixels.
To change, drag and drop a new file on top.

Name *

The service name should be a short text that uniquely identifies your service.

Short description *

This short description will be shown along with the name when your service is included in a list of catalog entries, so it should be less than a dozen words long, and be meaningful to a novice user who sees only the name, icon, and this text.

Long description *

This longer description can be several paragraphs long if you prefer. It will be shown in full when users view the details of your service.

Developer name *

Tags (Wikipedia categories)

The optional tags for this service will help a user discover your service in the catalog. Tags must be Wikipedia categories, must be separated by commas, and should be chosen to reflect both broader areas and narrower focus, for example a catalog entry may be in: "Statistics, Clustering". Each catalog entry may be in at most 10 tags.

Notice

Since this text aids later discovery it's worth taking some time to make sure it's nicely descriptive. You can further encourage discovery by adding category tags for your service. To encourage consistency among service developers we recommend using the names of Wikipedia categories rather than making up your own terms. In common with Wikipedia, these category tags need not be mutually exclusive. It is good practice to add both broad and narrow category tags for each service.

Having completed editing, it's time to review the entry as it will appear in the catalog:

The screenshot shows the 'Review details' tab for the 'Polite Greeting Service 1.0'. At the top, there are navigation buttons for '< Previous' and 'Next >'. Below that is a breadcrumb trail: 'Edit composition > Edit parameters > Edit description > Review details > Review code > Test > Publish'. A message states: 'This is how the details for your service will appear in the catalog, use the Edit Description tab to make any changes.' Below this is a diagram of the service with two input fields: 'Name' (string) and 'Greeting' (string). To the right of the diagram, it says: 'This composed service uses an input name to generate a personalized salutation.' Below the diagram, the inputs are listed: 'Name string' with the description 'The name of a person to be greeted' and 'Greeting string' with the description 'An uplifting salutation'.

And review the automatically-generated example code:

The screenshot shows the 'Review code' tab for the 'Polite Greeting Service 1.0'. At the top, there are navigation buttons for '< Previous' and 'Next >'. Below that is a breadcrumb trail: 'Edit composition > Edit parameters > Edit description > Review details > Review code > Test > Publish'. A message states: 'These instructions for calling your service were automatically generated from the description. You can use the Edit Description tab to make any changes. This code won't actually work until you publish your service.' Below this, it says: 'Here you can find some example code for calling this service from Java and JavaScript applications.' The 'JavaScript Code' section is highlighted, with the text: 'This uses jQuery to simplify the AJAX call.' Below this is a code block with the following JavaScript code:

```
1
2 // Construct the task header (which depends on the service being called)
3 var taskHeader = {
4   "mode": "synchronous",
5   "services": [{"id":"composition_14268798502523672426605", "version":"1.0", "engine":"Hybrid"}]
6   //, "timeout": 2.5 // This is an optional longer timeout in seconds in case your service will take a long time
7 };
8
9 // Construct the task data containing your particular input and output values
10 var taskData = {
11   // inputs and outputs here, e.g.:
12   "inputName": "CLASP User"
13 };
14
15 // Build the URL
16 var taskURL = "https://clasp.hpl.hp.com/ccfe/queue/1.0?" +
17   "taskHeader=" + encodeURIComponent(JSON.stringify(taskHeader)) + "&" +
18   "taskData=" + encodeURIComponent(JSON.stringify(taskData));
19
20 // And then retrieve it
21 // This used the jQuery ajax call (you may need a POST if the task data is large)
22 $.ajax({
23   type: "GET",
24   url: taskURL,
25   contentType: "application/json",
26   dataType: "json",
27   cache: false,
28   error: function() { alert("Something failed"); },
29   success: function(jsonStatus) {
30     alert("success!" + JSON.stringify(jsonStatus));
31   }
32 });
33
```

The 'Java Code' section is also visible, with the text: 'Add imports and exception handling appropriately.' Below this is a code block with the following Java code:

```
1
2 // Construct the task header (which depends on the service being called)
3 String taskHeader = "(" +
4   "\"mode\": \"synchronous\",\" +
5   "\"services\": [{"id\":\"composition_14268798502523672426605\", \"version\":\"1.0\", \"engine\":\"Hybrid\"}]\" +
```

Any imperfections in the reviewed content can be corrected by returning to preceding editing steps and adjusting the parameters and descriptions. Your service is saved each time you move to the next step, so you can always exit and resume the next day, taking your time to produce a nice catalog entry.

Prior to publishing it's necessary to test the service and demonstrate successful execution. This can be done simply using the default wizard:

Add/Edit Catalog Entries / Polite Greeting Service 1.0 < Previous Next

Edit composition > Edit parameters > Edit description > Review details > Review code > Test > Publish

This interface to support testing and evaluation of your service was automatically generated from the description. You must verify it works by testing successfully at least once before publishing.

You can try this service now using our data if you wish. If you're new to the system, then try the default wizard first, it will guide you through the process.

Default Wizard
Walks you through the process of testing the service.

1 Provide inputs values > 2 View result

Name *

The name of a person to be greeted

Submit

Raw I/O
Provides visibility into the data values and is most useful for testing settings for later use when calling the service from an application.

Notice that the wizard uses the names, descriptions, and any default values previously entered while editing parameters. In this case there was an additional input inside the composition, but since it's a constant it is not visible in the interface.

After execution, any resulting output (or execution error) is presented:

Default Wizard
Walks you through the process of testing the service.

1 Provide inputs values > 2 View result

Task Id: 7d4f5011-d410-44db-bb05-1653ee8563f5
Your task has been processed.

Greeting

Do a new test

More detail on the execution, including the execution logs, is available using the Raw I/O interface:

Raw I/O
Provides visibility into the data values and is most useful for testing settings for later use when calling the service from an application.

Name *
CLASP User
The name of a person to be greeted

Polite Greeting Service
Name string
Greeting string

Greeting
An uplifting salutation

Submit

Task 4a0486b4-cb1e-496b-b8a8-bb41384214d0 PROCESSED

Name *
CLASP User
The name of a person to be greeted

Polite Greeting Service
Name string
Greeting string

Greeting
Hello CLASP User, have a nice day!
An uplifting salutation

Execution log	No error	No warning	<input checked="" type="checkbox"/> 4 info	No debug
I _____	INFO		task 'Polite Greeting Service' execution	
I _____	INFO		task 'Hello World Service' execution	
I _____	INFO		The Hello World service is running	
I _____	INFO		task 'Concatenation Service' execution	
0 seconds 1	Total execution time 0.009 seconds			

The logs include an execution time graph. Note that during the very first invocation on each back-end server the system performs additional work retrieving the service descriptions, and since those are cached, later runs will generally be faster than the very first.

Once the service has been successfully tested, you can move to the final step of publishing:

Add/Edit Catalog Entries / Craig Sayers Composed Hello World Service 1.0 [← Previous](#)

[Edit composition](#) > [Edit parameters](#) > [Edit description](#) > [Review details](#) > [Review code](#) > [Test](#) > **Publish**

This service is now available for publication. Once published in the catalog, other users may view and then immediately start using your service. You can deprecate and then remove the service later if necessary.

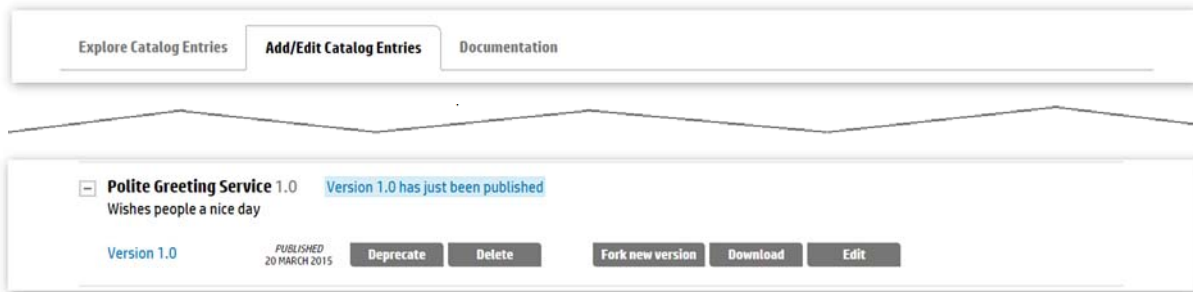
After Publication

Allow users to view inside this service

Allow this service to be used as a starting point for creating new services

Publish

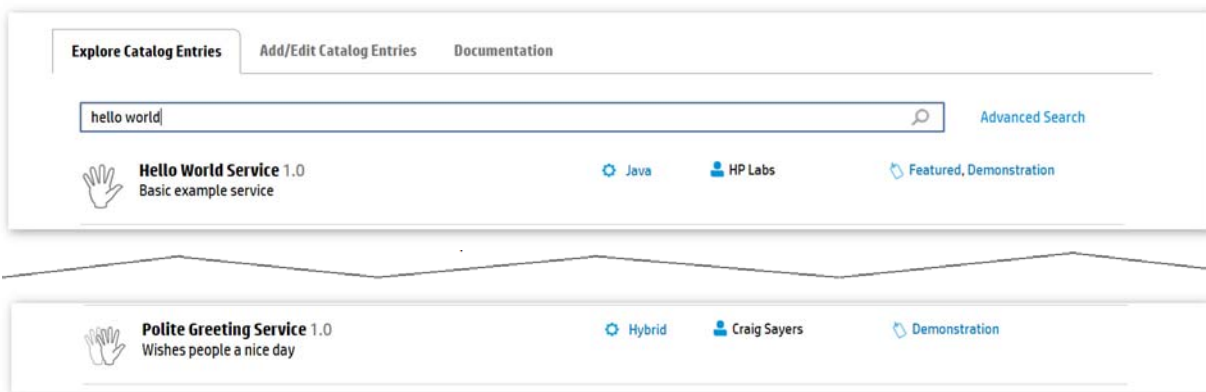
Services you have written will be visible in your add/edit catalog entries tab:



where you can control the service lifecycle, forking new versions while deprecating and ultimately deleting older versions. You can save services prior to the final publishing step, and at that point they are still private, so you can save intermediate results and take the time to iterate through several edit/test cycles before committing to a public service.

While you can also edit a previously-published service, this is only intended to correct minor errors in the human-readable descriptions. Remember that external developers may be using your service, so if you need to change the interfaces, then you should make a new version.

Your published services will be discoverable via search inside the catalog:



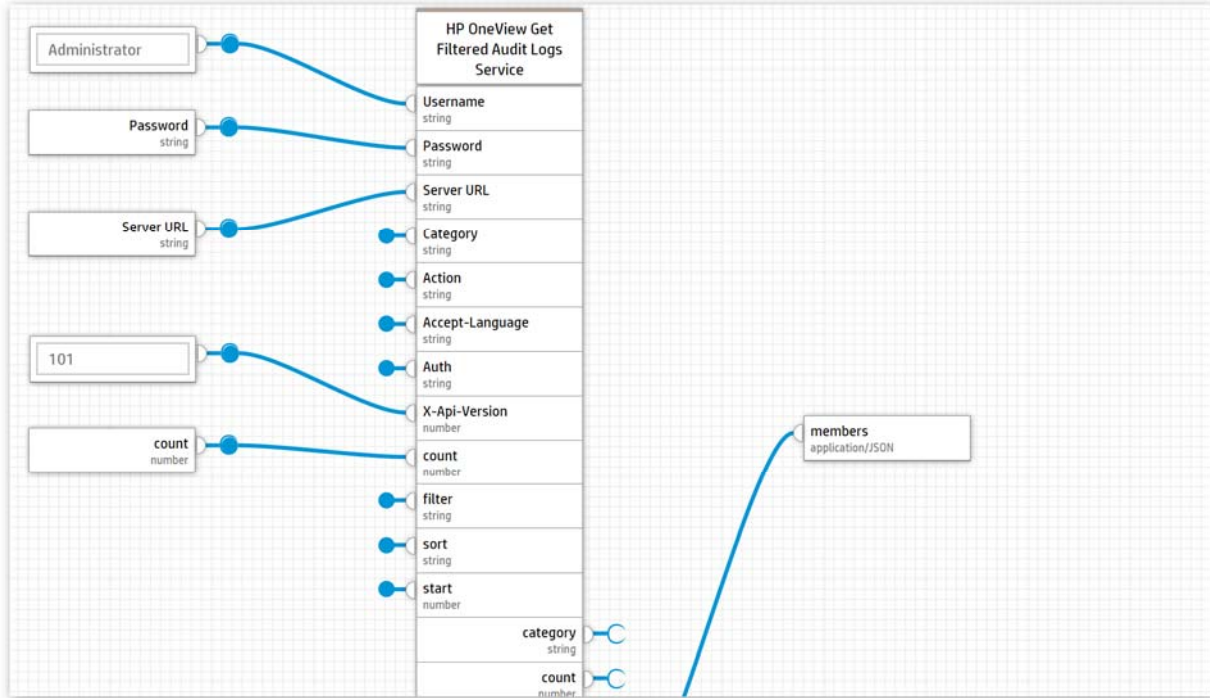
And they can then be viewed, tried, and composed in exactly the same way as any other service:



In this example we've composed multiple services together, but compositions can also be useful even for individual services, allowing you to hide many of the inputs and outputs, set them to be constants, and/or alter parameter names and descriptions to create custom domain-specific services.

The preceding example was deliberately simple but it's not uncommon for compositions to vary in size and complexity.

For example, consider an existing HP OneView operation for retrieving server logs. This has a number of input parameters, providing opportunities for customization, and is typical of APIs where it's generally simpler and preferable to have a smaller number of API calls, each of which is customizable, than a much larger number, each of which is more specialized. Using the compositional interface, we can take that general-purpose service and wrap it in a composition:



Making some of the inputs constant and removing many parameters creates a service which is more specialized, allowing it to be much simpler than the original:

Explore / Simple HP OneView Get Audit Logs Service 1.0

Simple HP OneView Get Audit Logs Service 1.0

Retrieves audit logs from the appliance

Summary

Details

Try

Call

Compose

View Inside

Simple HP OneView Get Audit Logs Service

Password string

Server URL string

count number

members application/JSON

Gets audit logs that are present in the appliance. This is a thin wrapper around the standard OneView operation to get filtered audit logs. Having fewer input and output parameters it's simpler to use, albeit less powerful.

Password *string*
Password for the Administrator user on the HP OneView system.

Server URL *string*
Url to your HP OneView system. The default url corresponds to a test system.

count *number*
The number of resources to return. A count of -1 requests all the items. The actual number of items in the [See more...](#)

members *application/JSON*
The array of resources contained in the specified collection.

Share

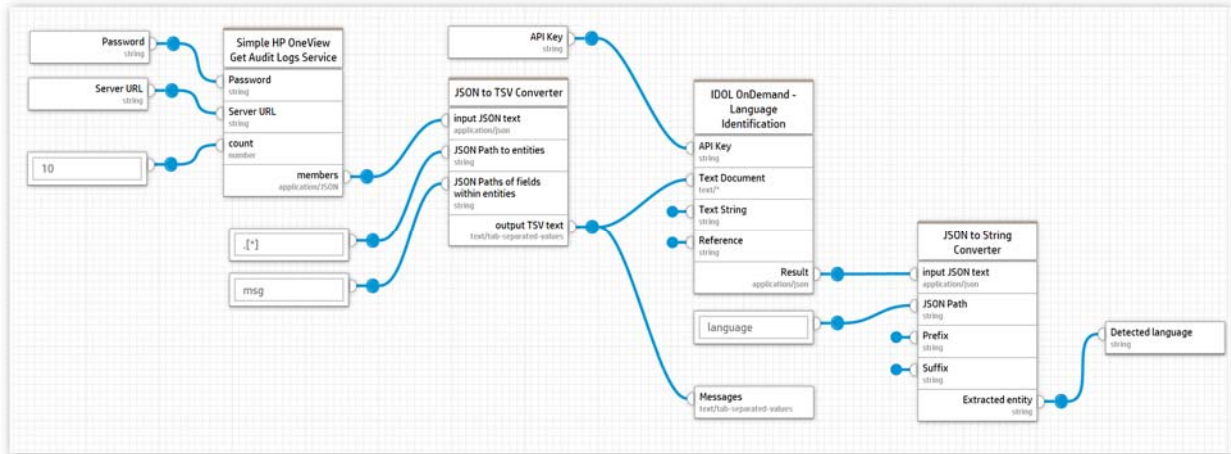
Developer
Craig Sayers

Tags
OneView
Activity
Hybrid

Related Services
Hello World Service
IDOL OnDemand - Sentiment Analysis
IDOL OnDemand - Face Detection

There could be many such services in the catalog, each customized for a particular application domain.

In this second example, we're combining the simplified service we just constructed with an Autonomy IDOL onDemand service to identify human languages in order to determine the language in which log messages are being written. This combines two very different pre-existing APIs with two internal services for extracting JSON content.



And that composition similarly appears in the catalog just as any other service:

Explore / Simple HP OneView Get Audit Logs Service 1.0 / Search / HP OneView Detect A...

HP OneView Detect Audit Logs Language 1.0

Gets filtered audit logs and detects the language used for the messages

Summary
Details
Try
Call
Compose
View Inside

HP OneView Detect Audit Logs Language

Password *string*
Server URL *string*
API Key *string*

Detected language *string*
Messages *text/tab-separated-values*

Password *string*
Password for the Administrator user on the HP OneView system.

Server URL *string*
Url to your HP OneView system. The default url corresponds to a test system.

API Key *string*
The API key for access to the IDOL OnDemand services. You can obtain an API Key for your own use from [See more...](#)

Detected language *string*
The language in which the log messages were written

Messages *text/tab-separated-values*
The extracted messages from the log

This composed service combines the OneView ability to obtain server logs with IDOL onDemand text processing to determine the language used for logged comments.

Developer
Craig Sayers

Tags
OneView
Activity
Hybrid

Related Services
Hello World Service
IDOL OnDemand - Sentiment Analysis
IDOL OnDemand - Face Detection

Share

Composition is the easiest way to make additional services, and is most appropriate when the component parts fit together smoothly. Of course that won't always be possible, and for those situations we encourage you to add new services by writing code.

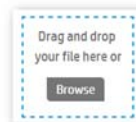
Service Development Using R, Python, JavaScript, or Java

For service development, our goal is to streamline the process. Developers are already familiar with local development tools using languages like Java, Python, R, and JavaScript, and so we leverage that, allowing local coding and debugging, while automating almost all the additional work for deployment as cloud services.

For example, if you have a local script such as this example which runs in R on Windows:

```
centers<-3.0; # input
x<-read.table(file='C: inputFile',header=TRUE,sep="\t"); x<-as.matrix(x);
library(stats);
function_Output=kmeans(x,centers);
cluster=function_Output$cluster;
write.table(as.matrix(cluster),file="C: outputFile",row.names=FALSE,col.names=TRUE,sep=",",quote=FALSE);
```

Then you can add that to the catalog by simply dragging/dropping to the box on the add/edit catalog entries tab. The system will then inspect the script, automatically extracting the inputs and outputs along with their corresponding types to generate a service.



The result looks like this:

Add/Edit Catalog Entries / Craig Sayers R Service 1.0

Save and Exit Undo < Previous Next >

Load files > Edit parameters > Edit description > Review details > Review code > Test > Publish

To prepare for publishing, make any necessary edits in the parameter descriptions and then click Next to proceed.

Inputs

- + centers (centers) number
- + x (x) R-Matrix

Add input

Operation

```
x<-as.matrix(x);
library(stats);
function_Output=kmeans(x,centers);
cluster=function_Output$cluster;
```

Edit

Outputs

- + cluster (cluster) R-Matrix

Add output

Notice that in the process we have removed the file I/O statements specific to the local installation, abstracting them as inputs and outputs. The 'centers' variable would have been an internal constant, but the

presence of the following `# input` comment in the ingested script provided sufficient additional information to make it an input also.

When the script is executed, the system will add appropriate statements to set/open/close any input or output parameters. Within your script you can just use any input variable as though it had been set (for elementary types) or opened (for more complex mime types), and similarly for output variables, you can just set the value (for elementary types) or write and close (for more complex mime types).

For the R language there are additional R-specific types of R-Vector, R-Matrix, and R-Dataframe. For compatibility with services in other languages, the Vector is equivalent to a text stream, where each line of the text is one entry in the vector; the Matrix is equivalent to a comma-separated-values (CSV) stream, where each line is one row, and columns are separated by commas; and the Dataframe is equivalent to a tab-separated-values (TSV) stream, where each line is one row, the first row contains column names, and columns are separated by tabs.

After verifying/correcting the initial ingestion, the subsequent steps for publishing are identical to those for the composite service described in the preceding section.

In addition to R, the catalog also supports Python and JavaScript services via a similar mechanism of developing and testing a local script, then dragging and dropping to the catalog. In those languages it is more challenging to automatically determine types, so it's helpful if you can mark any input or output variables with a comment to assist the system in adding them to the service catalog description.

For Python, it is not uncommon to have additional functions in an EGG file. In that case just create a .zip file containing your script and any EGG files and then drag/drop just as you would have done with the basic script.

The compiled nature of Java necessitates a little more complexity, but we aim to simplify the process. Developers download our SDK and after importing one of the included projects into Eclipse, can modify the code and debug/test within the Eclipse environment. Once satisfied, run the included build.xml Ant Script to create a service.zip file containing the .jar for your code and a description.json to describe the I/O parameters. That can then be dragged/dropped to the catalog just as the R script described previously, with subsequent publication following the same process as that for any other language or composed service.

As an example, the code for the Hello World Java Service (including some gratuitous logging) looks like this:

In describing parameters for local execution it is normal to use labels like “Input Matrix URL”, or “Output Cluster File”. However, in CLASP the transport is handled for you and is not known until runtime. For example, if your service is used in a composition then the input may be streamed from some preceding service, and even if called independently the input may come from a cloud storage location, or a database. Since your parameter name will be visible in all those cases, it's preferable to name parameters based on what they are, rather than how they are transported. Similarly in the parameter descriptions, don't say “The URL to your matrix”, instead take advantage of the opportunity to provide more detail about the expected input. At runtime CLASP will automatically add appropriate additional UI embellishments in the wizards aiding users in dragging/dropping local files or entering URLs; while in the compositional interface your names and descriptions will be key to aiding users creating compositions which are functionally correct.

```

public class HelloWorld extends Service {
    public void execute(Task task) {
        task.getLogger().info("The Hello World service is running");
        String in = task.getString("inputName");
        task.putString("outputMessage", "Hello " + in);
    }
}

```

The description.json file looks like this:

```

{
    "engine": "Java",
    "className": "com.hp.clasp.example.helloWorld.HelloWorld",
    "inputs": {
        "inputName": {
            "name": {"en": "Input name"},
            "type": "string"
        }
    },
    "outputs": {
        "outputMessage": {
            "name": {"en": "Output message"},
            "type": "string"
        }
    }
}

```

And you can test locally by constructing a sample input task.json file:

```

{
    "inputName": "World"
}

```

and then running with:

```

HelloWorld helloWorld = new HelloWorld();
System.out.println("The result is " + helloWorld.testService("description.json", "task.json"));

```

One subtlety is that services are run on many different back-end servers. So when writing your service it's important to avoid using any static variables, since those won't be shared among all instances of your service.

The Task object passed to your Service's execute method provides access to all the input and output parameters while providing a task-specific logger for information and debugging purposes. There is also an exception mechanism for handling reporting errors.

It may be tempting to return service status by adding additional parameters, however it is preferable to throw a TaskFailedException, allowing the system to know an error occurred and respond appropriately by stopping any composite execution and returning an appropriate error along with an error log to the user. For scripted languages, you can write messages to stderr and exit with a failure code (just as you would if running from the command line).

Here's a larger example of an image processing service:

```
public void execute(Task task) {  
  
    // Extract the width and height from the task inputs. If any required parameter is missing or invalid  
    // then CLASP will automatically generate an exception and the task will fail.  
    int width = task.getInt("targetWidth");  
    int height = task.getInt("targetHeight");  
  
    // Check for any obvious errors and problems in the input parameters  
    if (width < 2 || height < 2)  
        throw new TaskFailedException("Width/height are too small");  
    if (width * height > 4000000)  
        throw new TaskFailedException("Width/height are too large");  
  
    // Try and perform the image scaling  
    try {  
        BufferedImage inputImage = task.getBufferedImage("sourceImage");  
        task.getLogger().info("Starting Image scaling");  
        BufferedImage scaledImage = Scalr.resize(inputImage, Scalr.Mode.FIT_EXACT, width, height);  
        task.getLogger().info("Finished Image scaling");  
        task.putBufferedImage("outputImage", scaledImage);  
    } catch (Exception e) {  
        throw new TaskFailedException("Unable to process image", e);  
    }  
}
```

Notice the error check and exception mechanism to verify the input values, where in this case the validity of the maximum image size depends on a combination of width and height. This is not uncommon, and so while basic checks happen at the CLASP level (for example a missing required parameter, or incompatible type), more sophisticated checks are best achieved via code in the service.

If you have an existing large application, it is tempting to just wrap the entire thing as one service. While that may work, the inaccessibility of internal components restricts reusability. Rather than focusing on your own user community, take full advantage of the catalog by making your work available as a set of smaller services which can then be reused and combined by others in unanticipated, innovative, and frequently quite surprising ways.

Dataset Ingestion

Adding datasets to the catalog is convenient both for your own development needs and also to aid other users. This process is consistent with adding any other entry to the catalog.

The first step is to collect the data. This should be in a tab-separated-values (tsv) file and preferably have headings in the first row. For example, if we start with a toy example dataset:

Place	Average	Maximum
A	60	65
B	60	65
C	35	42
D	30	40

It can be added to the catalog by dragging/dropping that file using the add/edit catalog entries tab. The data is then analyzed, and a dataset catalog entry is created and populated with default human-readable descriptions:

1. Select data to be stored in the catalog *

None
In this case the dataset is only used internally by services, and the catalog entry assists users browsing the catalog.

Definition and example data values
In this case the dataset needs to be described so other service developers can use this dataset.

Definition and complete data values
In this case the complete dataset is loaded, stored, and managed by the catalog.

2. Provide a URL for your data *

Specify a URL for your input data.

3. Verify uploaded data

First row contains: *

Column names Data

1	2	3
Place	Average	Maximum
A	60	65
B	35	42
C	30	40
...

4. Describe each column in the dataset

	Column ID *	Name *	Type	Description
1	PLACE	Place	String	For example C.
2	AVERAGE	Average	Number	For example 30.
3	MAXIMUM	Maximum	Number	For example 40.


Notice that the column names are prepopulated using headings from the uploaded data, while draft content for types and descriptions are extracted from data analysis. You can edit these as appropriate to correct or embellish the entries. As for other publication, the goal is to encourage discovery and reuse within the catalog, so the better your descriptions, the more chance of that happening.

Subsequent steps are as for publishing any catalog entry. First editing the human-readable description:

Add/Edit Catalog Entries / Craig Sayers Test Dataset 1.0 Save and Exit Undo < Previous Next >

[Load files](#) > [Edit data](#) > [Edit description](#) > [Review details](#) > [Publish](#)

To prepare for publishing, make any necessary Edits, then Review the catalog entry.



icon.png
Uploaded 4:42pm 20 March 2015
Icons must be a PNG file and we recommend 256x256 pixels.
To change, drag and drop a new file on top.

Name *

The dataset name should be a short text that uniquely identifies your service.

Short description *

This short description will be shown along with the name when your dataset is included in a list of catalog entries, so it should be less than a dozen words long, and be meaningful to a novice user who sees only the name, icon, and this text.

Long description *

This longer description can be several paragraphs long if you prefer. It will be shown in full when users view the details of your dataset.

Developer name *

Tags (Wikipedia categories)

Then reviewing the catalog entry:

Add/Edit Catalog Entries / Craig Sayers Test Dataset 1.0 < Previous Next >

[Load files](#) > [Edit data](#) > [Edit description](#) > [Review details](#) > [Publish](#)

This is how the details for your dataset will appear in the catalog, use the Edit Description tab to make any changes.

A tiny dataset with synthetic data to demonstrate the ingestion of new datasets into the catalog.

Example Data

Place	Average	Maximum
PLACE (string)	AVERAGE (number)	MAXIMUM (number)
A	60	65
B	35	42
C	30	40
...

- 1 Place** *string*
For example C.
- 2 Average** *number*
For example 30.
- 3 Maximum** *number*
For example 40.

This dataset is an example for documentation purposes.

Before publishing:

This screenshot shows the 'Add/Edit Catalog Entries / Craig Sayers Test Dataset 1.0' page. At the top right is a '< Previous' button. Below it is a blue navigation bar with links: 'Load files > Edit data > Edit description > Review details > Publish'. A light blue box contains the text: 'This Dataset will be stored before publishing and this may take a while.' Below this, a paragraph states: 'This dataset is now available for publication. Once published in the catalog, other users may view and then immediately start using your dataset. You can deprecate and then remove the dataset later if necessary.' A red 'Publish' button is located at the bottom right.

After publication, you data persists in a CLASP back-end database, and your dataset is visible in the catalog:

This screenshot shows the 'Explore / Craig Sayers Test Dataset 1.0' page. The title is 'Craig Sayers Test Dataset 1.0' with the subtitle 'Example dataset with synthetic data'. A 'Share' button is in the top right. The developer is listed as 'Craig Sayers'. The description reads: 'A tiny dataset with synthetic data to demonstrate the ingestion of new datasets into the catalog.' Below this is a table titled 'Example Data':

Place	Average	Maximum
PLACE (string)	AVERAGE (number)	MAXIMUM (number)
A	60	65
B	35	42
C	30	40
...

Below the table is a numbered list of field descriptions:

- Place** *string*
For example C.
- Average** *number*
For example 30.
- Maximum** *number*
For example 40.

The text 'This dataset is an example for documentation purposes.' follows. Below that, it says 'Access this dataset directly using these services:' and lists a 'Query Craig Sayers Test Dataset 1.0' service with the description 'Select from the Vertica table that represents a dataset.' and a link to 'Vertica, DatasetQuery, Hybrid'.

An additional query service will have been automatically constructed:

Explore / Craig Sayers Test Dataset 1.0

Craig Sayers Test Dataset 1.0

Example dataset with synthetic data

Share

Developer
Craig Sayers

Tags
Dataset

Details

Try

Try accessing this dataset directly using these services:

Query Craig Sayers Test Dataset 1.0
Select from the Vertica table that represents a dataset.

Vertica, DatasetQuery, Hybrid

and can be employed to extract data using SQL:

Query Craig Sayers Test Dataset 1.0

Select from the Vertica table that represents a dataset.

Share

Developer
HP Labs

Tags
Vertica
DatasetQuery
Hybrid

Related Services
Hello World Service
IDOL OnDemand - Sentiment Analysis
IDOL OnDemand - Face Detection

Summary

Details

Try

Call

Compose

You can try this service now using our data if you wish. If you're new to the system, then try the default wizard first, it will guide you through the process.

Default Wizard
Walks you through the process of testing the service.

1 Provide inputs values > **2 View result**

Input string *

select * from %TABLE% limit 3

Input string to perform search and replace on.

Submit

Just as for any service output, the result can be previewed in the browser:

Query Craig Sayers Test Dataset 1.0

Select from the Vertica table that represents a dataset.

Share

Developer
HP Labs

Tags
Vertica
DatasetQuery
Hybrid

Related Services
Hello World Service
IDOL OnDemand - Sentiment Analysis
IDOL OnDemand - Face Detection

Summary

Details

Try

Call

Compose

You can try this service now using our data if you wish. If you're new to the system, then try the default wizard first, it will guide you through the process.

Default Wizard
Walks you through the process of testing the service.

1 Provide inputs values > **2 View result**

Task Id: 21bf1966-63d6-474a-b9f9-f94de5dfb6fb
Your task has been processed.

Result

<https://clasp.hpl.hp.com/ccfe/results/tasks/p2/21bf1966-63d6-474a-b9f9-f94de5dfb6fb.queryVertica1/output.tsv> Hide Preview

PLACE	AVERAGE	MAXIMUM
A	60	65
B	60	65

and employed as a starting point for compositions, just as any other service.

A number of example datasets are pre-installed in the system. The following table lists some examples. Just search the Datasets category tag to find these and others in the catalog:

AIRPORT	Airport name, code, city and latitude-longitude information
BALTIMORE CITY EMPLOYEE SALARY 2014	Baltimore city employee salaries 2014
CALIFORNIA INCOME TAX	Income tax returns for state of California by county for tax year 2009
EURO FOREIGN EXCHANGE RATE	Euro daily foreign exchange rate from January 4,1999 to April 8, 2014
IRIS	Fisher's Iris flower data
NEW YORK BABY NAMES	New York region baby names from 2007 to 2012
NEW YORK HOSPITAL	New York hospital cardiac surgery from 2008 to 2010
SEATTLE TECHNOLOGY FUNDS	Technology projects in Seattle that have been awarded funds from 1998 to 2013
SAN FRANCISCO BICYCLE PARKING	San Francisco city public bicycle parking locations
SAN FRANCISCO HOUSE PRICES	San Francisco metropolitan area monthly house prices from December, 1986 to April, 2013
WORLD DISASTERS	World disasters from 1900 to 2008
US PUBLIC LIBRARY	US public library survey for fiscal year 2011

In these datasets, the data is actually contained within our system. A key competitive advantage of many service developers is the underlying data on which their service depends. Since they may be unable or reluctant to give that away, we also needed a mechanism to advertise usage via their services. That's handled via the special-case of Reference datasets. These don't include any data directly, but can be discovered in the catalog, and they link to related services, serving as a navigational aid for users and a capability advertising mechanism for developers. Currently these are added to the catalog via manual editing/ingestion, but in the future will be supported via the catalog UI.

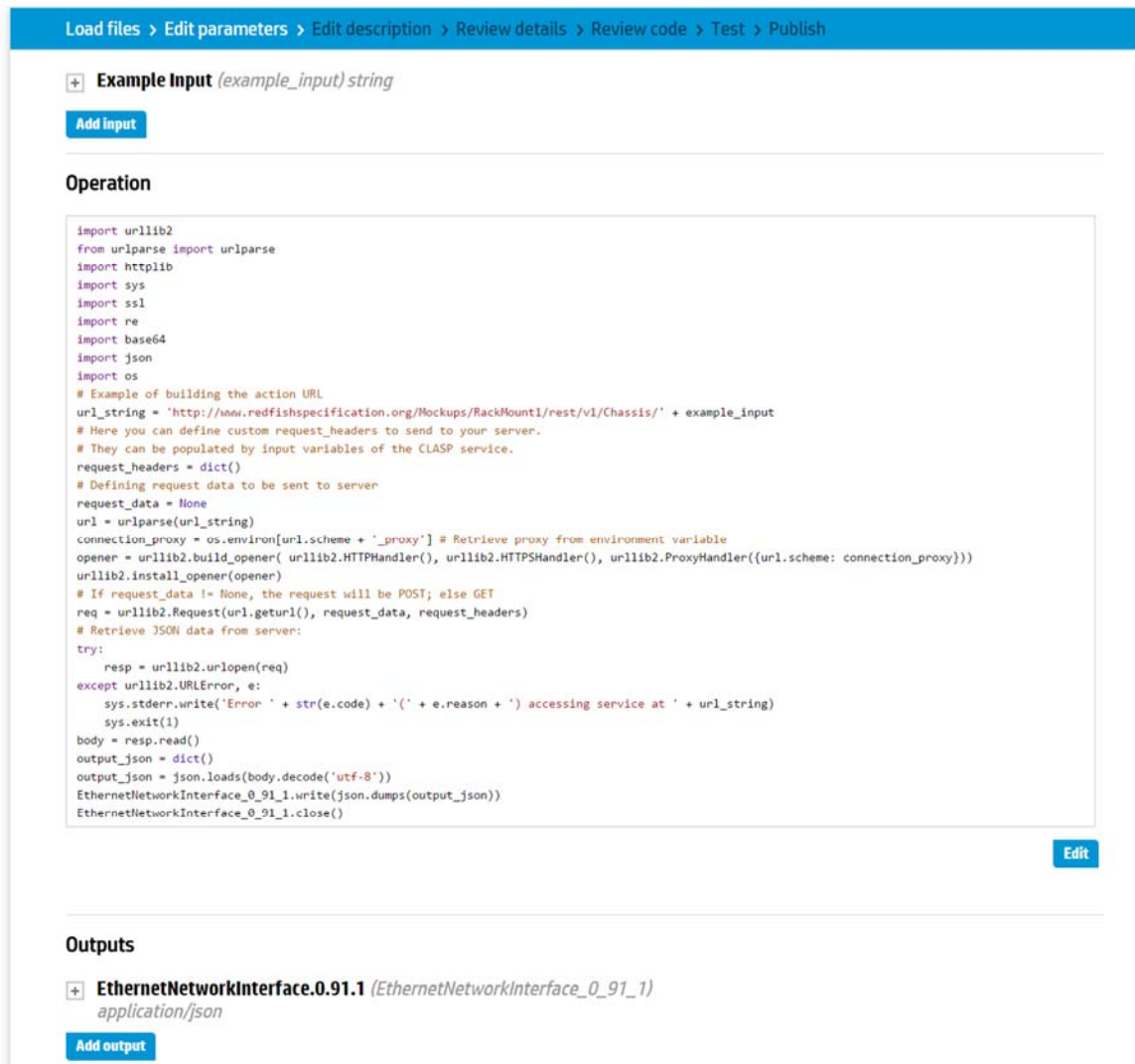
Accessing External APIs

While we would generally prefer developers rely entirely on our own system, we recognize both the mass of existing services and the need for some services to run on dedicated hardware. Thus we also support access to external web APIs from within our platform.

Custom services for external APIs

The general idea is to construct a thin wrapper which appears just as any other service inside our catalog. Inside that wrapper the CLASP inputs are converted to API-specific ones, the external API is called, and the results are then converted into CLASP outputs. These wrapper services are created exactly as any other Java or scripted service (though you may need to be an administrative user to add the service while bypassing the security checks, allowing external access from within a service).

An especially common case is for web services which generate JSON output, and so we have a semi-automated process for those. Simply drop the JSON Schema describing the output onto the add/edit catalog entries box (it must end in .json to be recognized), and we'll generate a scripted service using Python with a pre-populated output described by your schema, and an example input parameter:



The screenshot shows a web interface for configuring a service. At the top, a navigation bar contains links: Load files > Edit parameters > Edit description > Review details > Review code > Test > Publish. Below this, there is a section for "Example Input" with a sub-label "(example_input) string" and an "Add input" button. The main section is titled "Operation" and contains a Python script. The script imports various modules (urllib2, urlparse, urllib, sys, ssl, re, base64, json, os) and defines a function to build an action URL. It then constructs a request with headers and data, uses a proxy handler to access the service, and returns the JSON response. An "Edit" button is located at the bottom right of the code block. Below the code, there is an "Outputs" section with a sub-label "EthernetNetworkInterface.0.91.1 (EthernetNetworkInterface_0_91_1) application/json" and an "Add output" button.

```
import urllib2
from urlparse import urlparse
import urllib
import sys
import ssl
import re
import base64
import json
import os

# Example of building the action URL
url_string = 'http://www.redfishspecification.org/Mockups/RackMount1/rest/v1/Chassis/' + example_input
# Here you can define custom request_headers to send to your server.
# They can be populated by input variables of the CLASP service.
request_headers = dict()
# Defining request data to be sent to server
request_data = None
url = urlparse(url_string)
connection_proxy = os.environ[url.scheme + '_proxy'] # Retrieve proxy from environment variable
opener = urllib2.build_opener( urllib2.HTTPHandler(), urllib2.HTTPSHandler(), urllib2.ProxyHandler({url.scheme: connection_proxy}))
urllib2.install_opener(opener)
# If request_data != None, the request will be POST; else GET
req = urllib2.Request(url.geturl(), request_data, request_headers)
# Retrieve JSON data from server:
try:
    resp = urllib2.urlopen(req)
except urllib2.URLError, e:
    sys.stderr.write('Error ' + str(e.code) + '(' + e.reason + ') accessing service at ' + url_string)
    sys.exit(1)
body = resp.read()
output_json = dict()
output_json = json.loads(body.decode('utf-8'))
EthernetNetworkInterface_0_91_1.write(json.dumps(output_json))
EthernetNetworkInterface_0_91_1.close()
```

You can then edit that example to create your custom service, adding/modifying input parameters as necessary. In common with all scripted services, each input parameter will have its value set by CLASP before your code is executed. So in the above example the input 'example_input' can be used in the script.

The output parameter details will have been prepopulated using the provided schema, but you may wish to manually edit the name (extracted from the schema title) or description (extracted from the schema description) since the schema values for those can sometimes be a little terse.

After editing, the remaining steps to describe, test, and publish are consistent with any other catalog service.

Bulk Ingestion of services for external APIs

It's generally easiest to add individual services manually. However, when a large number of new services are needed, automation becomes necessary. This is particularly true when wrapping external APIs so they can appear as services within the catalog.

The recommended procedure is as follows:

1. Obtain the most structured and machine-readable form of documentation possible. It's not uncommon for html or pdf files to have been generated from some more structured format, and so worth asking if the source data is available.
2. Build a single service manually to exercise a representative API call. This will aid you in finding/fixing any issues with networking or permissions early, and this service will serve as a template for all the other services. It's usually easiest (though not necessarily fastest) to write this using Python as the scripting language. Once that's working, use the Download option for your service on the Add/Edit Catalog Entries page to obtain a copy.
3. Now write a script which mines the documentation files, generating one service description.json file for each API using your manual service as a template, and create a .zip file for each service including that generated description.json and an icon.
4. Try dragging/dropping several of those to the catalog to test they work and fix any issues by editing your generation script.
5. Now, to add all the services to the catalog, you have some choices. As an external user you can manually drag/drop each one – requiring only a few minutes per service, you could reasonably do over one hundred in a day. The downside is that you'll need to edit each service in order to make any subsequent change. If you have access to the servers running CLASP, you can generate a single .zip containing all of your service.zip files, and then add that to the installation directory for CLASP, editing the appropriate properties file to include that .zip file as one of the initial sources for all the system services in the catalog at startup time. In this way your service generation code can be checked into a source code repository and run as part of a standard build process.

Implementation and Infrastructure

The CLASP system handles both the catalog and task execution, where a single Task is one invocation of a Service (which may be composed of several component Services, which may themselves be composed, and so on).

Applications invoke Tasks directly using the Task Execution engine which pulls necessary service descriptions and code from the Catalog. These are cached, speeding subsequent calls to any service. Subsequent internal processing depends on the service language. For Java, the required .jar files are loaded from the service description using a custom class loader prior to invoking the execute method. For scripted services, a custom run-time version of the script is generated by combining the task inputs/outputs with code from the service description, and then executed via the appropriate language runtime. The services may directly access CLASP Datasets, and/or external data, or services.

Logical View

The Catalog is split between the client browser UI and the back-end to support that. Basic operations are to add and publish entries (persisted in the Catalog DB), search and display service details (querying the Catalog DB), and testing services (via requests to the Task Execution engine). For datasets, the Catalog additionally supports inserting data into the Dataset store, and querying for preview generation.

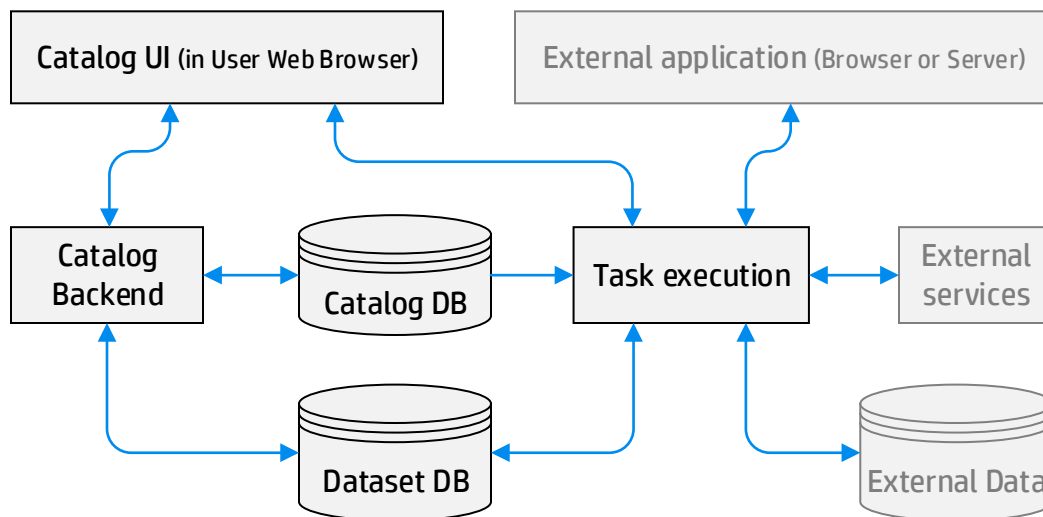


Figure 3. Logical view of the CLASP system

Physical View

The CLASP system physical implementation follows a regular web-server model, with a load balancer at the front, followed by several Tomcat servers to process servlet requests, backed by several databases providing persistence and consistency.

In current deployments the servlets for handling Catalog UI requests and those for processing Tasks (including all required languages/libraries) are present on every Tomcat server. This is an implementational convenience, allowing a single configuration. It should be possible to split those, having the UI on one set of servers behind one load balancer and the execution on another set of servers behind a second load balancer, allowing the UI to function smoothly even if the task execution servers became overloaded.

Databases hold all persistent state, permitting subsequent requests to be routed to any server, allowing additional Tomcat servers to be added at runtime, and limiting the effects of any machine failure to only current or recent operations performed on that particular server.

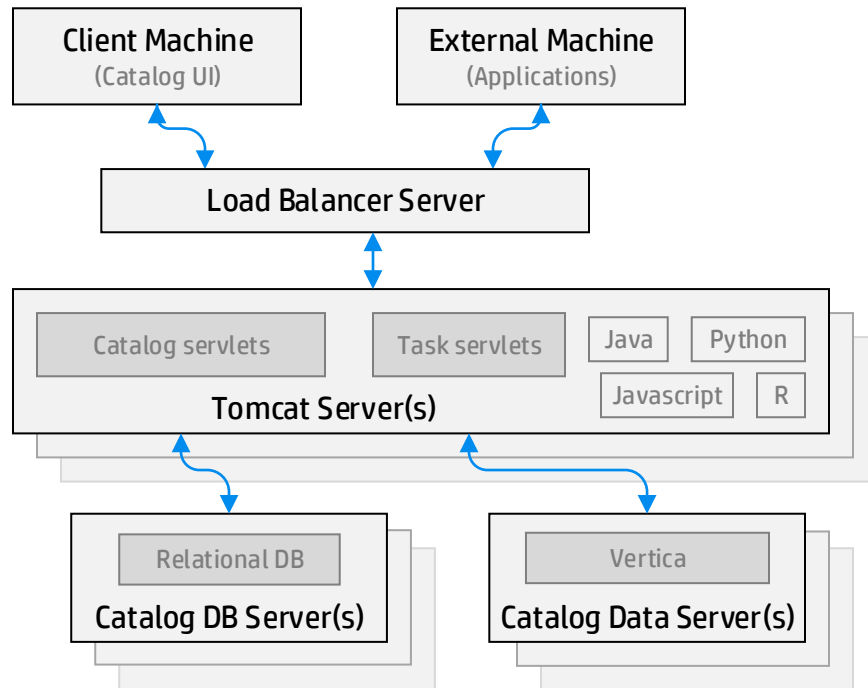


Figure 4. Physical View of the current CLASP system implementation

This figure corresponds to our current and recommended deployment. However if that level of redundancy and complexity is not required then much simpler installs are possible. We have, for example, installed the CLASP system on a laptop for demonstration purposes. In that case we can dispense with the load balancer and use single-node versions of the databases. Our nightly build server has a similar single-node configuration, running a full CLASP system updated nightly from the build as part of our testing process.

Security

Designed for deployment behind a corporate firewall, we have somewhat lower requirements for security than a public website. Nevertheless there are a number of security issues to consider.

For adding new services to the catalog we require users to login first. In our current implementation this is either by using LDAP, or by using an Autonomy API Key. The login requirement provides recognition (we show user's names alongside their services) but also provides attribution so any problematic service can be traced back to the original author and the users of such services know who to contact for assistance. User accounts within CLASP are created automatically upon the first login and our reliance on external authentication means we can avoid storing any passwords.

When running services, we are concerned not to accidentally retain any user-supplied information. Much of that comes as a convenient side-effect of the stateless nature of service execution, however we have also considered other ways information may leak and as a consequence have paid particular attention to logging. When a task is executed we generate a task-specific in-memory execution log and upon completion (either

successful or unsuccessful) we return that to the caller and remove any copy on our servers. Thus the logging from service execution is available to the caller, but not to the service developer.

We also limit the execution time for services. Since our developers are internal to the organization, this is not so much for malicious intent, but instead primarily to catch programming errors. The typical timeout is short (only a minute or so) but we have practical use-cases where longer times are needed, and we do allow longer timeouts to be requested during task invocation. Again, this is a case where we trust our users, but with some limit to catch errors.

For service ingestion, we add additional checks for security with the goal of limiting services from performing harmful operations. For scripted services we parse the script to find problematic commands such as `exec` or command-line invocations, and prevent the ingestion of those by non-administrative users. If the catalog were public, we would also have to lock down Java services – there are some techniques for limiting the available libraries and/or running inside sandboxes which may be helpful, but since our current users are all internal to the organization, and the system is within a firewall, we've been able to avoid adding that complexity. In anticipation it may be needed later, we do have the ability for services to make security assertions (which are shown to users, and may be verified programmatically) via the security property in the service description.json.

Speed

Execution speed is a perennial issue for all systems, and CLASP is no exception. We have thus expended some effort to quantify and reduce system overhead. Somewhat surprisingly, the logging system was our largest overhead, and we eliminated much of that cost by moving to a customized in-memory logger. A second cause of significant delay was a misconfigured proxy setting, and so for future installations we recommend checking that via the benchmark ByteCounter service in the catalog to verify correct configuration.

Since our services run inside a servlet and are accessed via HTTPS, we pay the cost of that overhead, but it's largely unavoidable while retaining common web standards and development usage patterns. On our system/network we see an overhead of around 24ms for calling any servlet via HTTPS. The additional overhead of CLASP services is around 2ms. For CLASP services which read content from external URLs (a particularly common case), the speed of ingestion is similar to using `wget` from the command-line at around 25ms for 10 Mb via HTTP from a local server (in practice it's much larger since the servers are typically much further apart, but that overhead is outside our control and again unavoidable). We also need to read data from databases, and we have similarly optimized that operation, so the speed for SELECT statements is now approximately the same as querying using standard command-line tools.

Note that the very first execution of each service on any server requires the server to load and cache the service description and any executable code, so all services will take as much as several hundred milliseconds longer the very first time. Also note that the servers are shared, and so may also be simultaneously processing tasks from other users.

Our internal deployment uses a load balancer and multiple servers, this adds redundancy and supports larger loads, but with the inevitable added cost of the additional network transactions.

The system operates in a web-server style, where many requests are processed simultaneously, thus the time to complete N tasks is typically substantially faster than the time-per-task multiplied by N. We have parallelization at the task level, with the expectation of there being many tasks and maximize throughput by minimizing the work required for each individual service.

Example performance measurements are shown in Figure 5 and Figure 6. In this case we're using a large number of threads to simultaneously fire requests at the CLASP system. Each of those threads simulates an application which sends a new request as soon as the response from its previous request is received. Using the simplest in our catalog (HelloWorld), it is a throughput test which should be limited by the overhead of CLASP and our servers rather than any service processing or I/O.

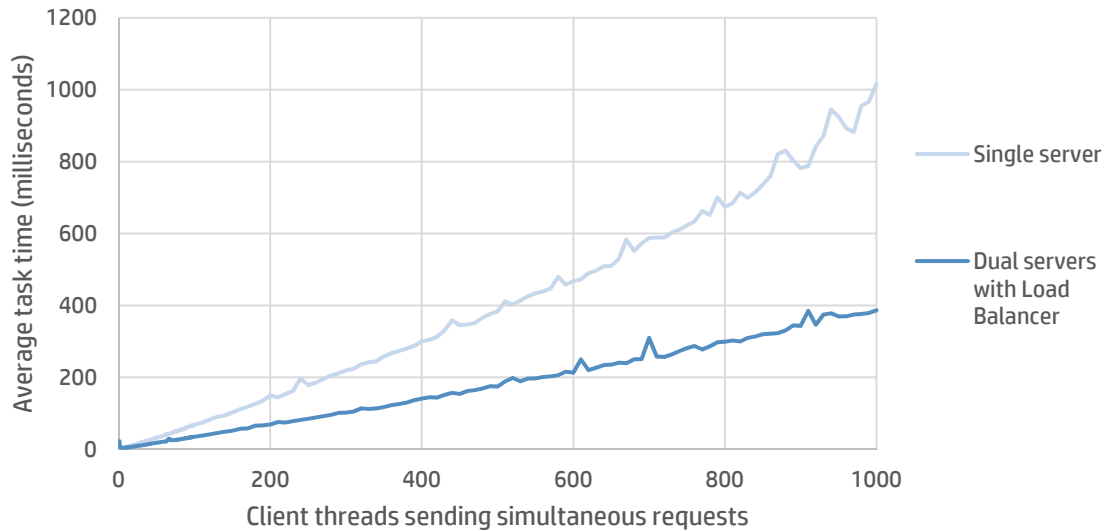


Figure 5. Task execution time (as measured by the client). Notice that even with 500 simultaneous threads (simulating 500 applications) the average processing time remains less than 200ms with dual servers.

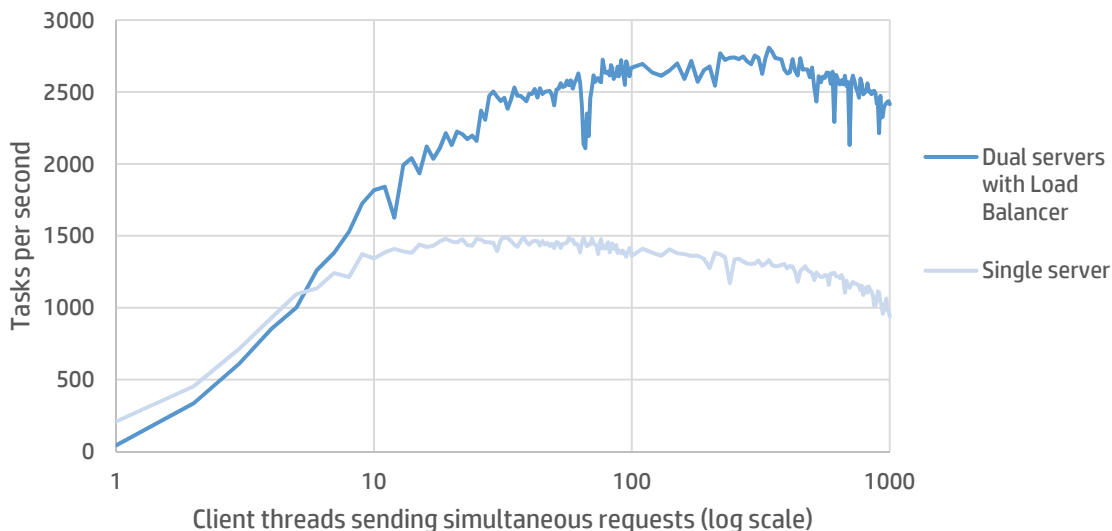


Figure 6. Tasks completed per second with up to 1000 client threads (each simulating an application). Once there are more than half-a-dozen simultaneous requests the load balancer overhead becomes insignificant and under heavy load the dual servers offer significantly better performance.

These results are not surprising, following a typical web server model, and demonstrate that our service execution doesn't impede that conventional scale-out behavior.

Service Definition

Services are defined by a JSON document which defines service operation along with the input and output parameters, and contains human-readable content for the catalog. In many cases service descriptions will be created automatically as you edit the corresponding catalog entry and so you need not be concerned with the internal details. However, with Java services you need a minimal `description.json` so you can run and test your service locally prior to uploading, and for all languages, if you choose to download your service it will include a `description.json` file. The description file has four mandatory fields, while the remainder are optional and if absent will typically be added by the catalog automatically when the service is ingested.

For Java services, the four mandatory fields are `engine` and `className` for the code, along with `parameter inputs` and `outputs`. So the smallest legal Java `description.json` is:

```
{
  "engine": "Java",
  "className": "com.hp.clasp.example.service",
  "inputs": {},
  "outputs": {}
}
```

For scripted services, the minimal service is similar except the code itself is included in the service description. For example, for R:

```
{
  "engine": "R",
  "RScript": "# R code goes here "
  "inputs": {},
  "outputs": {}
}
```

And similarly for Python and JavaScript – just replacing each occurrence of ‘R’ with the scripted language name.

Of course, to do something interesting, you’d need at least one input or output, these have similar definitions¹, with a typical example of an outputs definition being:

```
"outputs": {
  "msg": {
    "name": {"en": "Message"},
    "type": "string",
    "description": {"en": "An example output message"}
  }
}
```

This includes the machine-readable identifier (`msg`) and `type`, along with the human-readable content for the catalog (name and description) which in this case happens to be in English (`en`).

¹ At the time our JSON format was created, the JSON Schema was immature, it has since improved, and so it may be worth adjusting the specification in the future to use JSON Schemas to describe the inputs and outputs. We do have additional information not captured by the standard JSON Schema parameters, but it permits additional entries which could be used to support our needs. Note that we do already support the use of JSON Schema to describe individual input or output parameters with `type application/json`.

Common parameter definitions

The most commonly-used parameters are as follows:

- id** An identifier for your service. This must be unique across all services by all authors, and will be maintained by the catalog.
- version** The version number of this service. This will be updated and managed automatically by the catalog so that changes in versions can be associated with informational messages for catalog users.
- engine** The engine on which this service can run. For example 'R' for R scripted services.
- name** A short human-readable name for your service (just a few words). To avoid user confusion this must also be unique to your service.
- shortDescription** A short description of your service. This will be shown when your service is included in a list of services, so it should be meaningful to a novice user who sees only the name, icon, and this short description, but also relatively short.
- description** This is a longer description. It can be several paragraphs long if you prefer, and will be shown when a user views the details of your service.
- categories** This is an array of the category tags in which your service should be shown in the catalog. These are the Wikipedia categories such as Image Processing, Parallel Computing, Text Processing, etc. It is desirable to have both narrow and broad categories to aid future search operations.
- developerName** This is the name of the developer (either a person or a company). You need not list your own name here if you don't want to.
- developerURL** This is a unique URL for this developer. It is important to use the same URL for all services you develop. (In the future, we'll use this to arrange services by developer while browsing the service catalog.) If you don't have your own website, use an email address with a mailto transport, for example: `mailto:somePerson@someProvider`.
- support** This is the URL which users can visit to obtain support for this particular service. This should be a link to your own web page. Each service you develop can have a different support URL if you wish. Again, if you don't have your own website, use an email address with a mailto transport, for example: `mailto:somePerson@someProvider`.
- security** An assertion by the service developer that the service does not perform particular operations. Currently only used for Java applications, and only for a visual indication, it is intended as the basis for later more sophisticated and verifiable assertions. An example is: `{ "java.io":false }` indicating this service does not perform i/o operations other than via the CLASP-provided functions (and hence that it could not secretly transmit user data to an unknown party during execution).
- icon** This is a small image in PNG format with the name 'icon.png' that represents the service. It should be square and we recommend 256x256 pixels.
- notice** This text appears toward the end of the description in the displayed service details. It's a convenient place for any ancillary information such as acknowledgements.

bibliography This is the bibliography, allowing you to include references or footnotes in the human-readable descriptions. To use, first place a citation with the form: `\cite{†}` in any piece of the text (for example in the description property). And then within this bibliography parameter, include one property for each citation with the form: `"†": {"en": "some reference here"}`

In this example we've used the dagger symbol, but you can use any other symbol as well (for example `‡` or `§`), allowing disambiguation for several footnotes in a single service.

inputs and outputs These are the inputs and outputs for the service. These are JSON objects in which each attribute uniquely identifies one i/o parameter, and the value of that is a definition object containing:

name	A human-readable name for the input or output.
description	A human-readable description for this property to be displayed for developers. This is a good place to describe any limitations on the accepted values.
type	The type of this input or output parameter. This may be one of the scalar types: 'number', 'string', 'image', 'boolean'; or any MIME type, for example: 'text/plain', 'application/pdf', 'image/jpeg'; or a sequence of strings, images, or any MIME types, for example: 'sequence[string]', or 'sequence[image/jpeg]'. Wildcards are also supported in the MIME types, for example: 'image/*' is any image, while 'sequence[*/*]' is a sequence of streams whose type is unknown. In addition, we support R-specific types of R-Vector, R-Matrix, and R-Dataframe. An R-Vector can be automatically mapped to text/*, where each line is one entry in the vector. An R-Matrix, can be mapped to text/csv, where each line is one row, and each entry is the value for one element. An R-Dataframe can be mapped to text/tsv, where each line corresponds to one row, each entry to one element, and the first row has column names.
required	If true, the parameter must have a value in any input task, if false, the parameter need not be supplied.
schema	The optional schema of this input or output parameter. This is most commonly-used with parameters of type application/json to hold the JSON Schema describing the parameter value.
default	A default value for the parameter when displayed in a user interface. This is most useful for input parameters, and for non-scalar output parameters.
style	The optional style property only applies to inputs. It may be absent (implying a normal input), or set to password (implying the parameter is a password and changing the user interface display and managing cookies accordingly), or constant (in which case the value is always the default value, and, being supplied automatically to the service, it need not appear in the UI).

x and y The x and y location of this parameter in the displayed compositional editor. Rather than placing objects automatically (and frequently getting it wrong) we instead just remember where the user chose to place the parameter and then replace it in that same spot. In this way the view inside the editor remains consistent across editing sessions, and has spacing between parameters and components considered appropriate by the service creator.

Elementary service parameters

Elementary services (which are not compositions, but actually contain code) must have one of the following:

- className** This is used only for Java services, and is a full Java class name that that implements the service. When you create a Java service (see section "Service Development Using R, Python, Javascript, or Java"), you will create a ZIP file containing the description.json file, the service icon, and one or more JAR files with the required Java classes. The className parameter is the name of the class that extends the CLASP Service class and implements your service. In the HelloWorld example above, this would be "com.hp.clasp.example.HelloWorld".
- RScript** This is used only for R services, and contains the R code which implements the service.
- PythonScript** This is used only for Python services, and contains the Python code which implements the service.
- JavaScriptScript** This is used only for JavaScript services, and contains the JavaScript code which implements the service.
- PigScript** This is used only for Pig services, and contains the Pig code which implements the service. While we don't currently support Pig in our production system, we have used it in the past, and it's another example of a scripting language.

For each of the scripted services, you can generally take a conventional script which works on your local machine and then just drag/drop the file into the add service box in the catalog.

Java Hello World Example

Here's an example of the description for the Hello World example:

```
{
  "id": "abc.xyz.helloWorld",
  "version": "0.1",
  "engine": "Java",
  "className": "com.hp.clasp.example.helloWorld"
  "name": {"en": "Hello World Service"},
  "description": {"en": "This is a very basic example service which simply says Hello."},
  "shortDescription": {"en": "Basic example service"},
  "categories": ["miscellaneous"],
  "developerName": {"en": "HP Labs team"},
  "developerURL": "www.hp.com",
  "support": "TBD",
  "icon": "icon.png",
  "inputs": {
    "inputName": {
      "name": {"en": "input"},
      "required": true,
      "type": "string",
      "description": {"en": "Name"}
    }
  },
  "outputs": {
    "output": {
      "name": {"en": "output message"},
      "type": "string",
      "description": {"en": "The output salutation"}
    }
  }
}
```

Scripted Fast Fourier Transform Example

Here's an example description.json file for our automatically-generated R Fast Fourier Transform service:

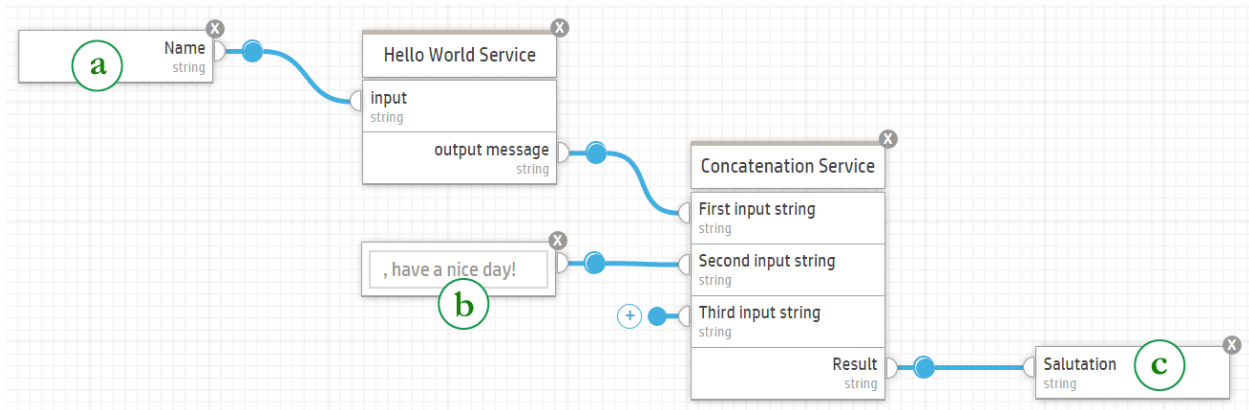
```
{
  "name": {"en": "R fft Service"},
  "id": "hpl-stats-fft",
  "version": "1.0",
  "shortDescription": {"en": "\Fast Discrete Fourier Transform\cite{&dagger;}"},
  "description": {"en": "\Performs the Fast Fourier Transform of an array.\cite{&dagger;} This is an automatically generated service that is currently at an experimental stage. Please let us know if you experience any problems when using this service."},
  "engine": "R",
  "categories": ["R", "Statistics", "RExt"],
  "developerName": {"en": "HP Labs"},
  "developerURL": "www.hp.com",
  "support": "TBD",
  "icon": "icon.png",
  "inputs": {
    "zInput": {
      "name": {"en": "z"},
      "required": true,
      "type": "R-Matrix",
      "default": "example:hpl-stats-fft-Matrix",
      "description": {"en": "a real or complex array containing the values to be transformed."},
    }
  },
  "outputs": {
    "fftOutput": {
      "name": {"en": "fftOutput"},
      "required": true,
      "type": "R-Dataframe",
      "default": "task:fftOutput",
      "description": {"en": "Output of the R fft service."},
    }
  },
  "RScript": "library(stats);\nlibrary(methods);\nlibrary(nlme);\n  function_Output=fft(z=zInput);\nfftOutput=function_Output\n",
  "notice": {"en": "When this service is executed it calls the R function fft(). The full documentation is available with R from <a href='\"http://www.r-project.org/\">http://www.r-project.org/</a>. R is free software and comes with ABSOLUTELY NO WARRANTY. R may be redistributed under the terms of the <a href='\"http://www.gnu.org/copyleft/gpl.html\">GNU General Public License</a>."},
  "bibliography": { "&dagger;": {"en": "The documentation for this service was created automatically based on the help file for this function. The full documentation is available with R from <a href='\"http://www.r-project.org/\">http://www.r-project.org/</a>. "}}
}
```

Composition Definition and Example

Composed services are described in the service description.json with a graph model, where component services and I/O parameters are the *nodes*, and connections between them are the *edges*. You can easily construct compositions graphically using the catalog UI, so the following description is only for those rare case where composed service descriptions are generated outside the interface (for example by using a script).

- components** An array of component services. For a composition, this must contain at least one service, each object in the array is one component service and has:
- id** A local ID for the service within this composition (a service may be included more than once, hence the need for an ID which is distinct from the ID of the service itself).
 - service** An object identifying the component service. This is a particular instance of a service with unique id and a defined version and engine so there is no ambiguity. There is exactly one service in the catalog with these three matching values:
 - id** the identifier for the component service (this is the id parameter at the top level of the service's description.json).
 - engine** the engine for the component service.
 - version** the version number for the component service.
 - x and y** The x and y location of this service in the displayed compositional editor. Rather than placing objects automatically (and frequently getting it wrong) we instead just remember where the user chose to place the service and then replace it in that same spot. In this way the view inside the editor remains consistent across editing sessions, and has spacing between components considered appropriate by the service creator.
- connections** An array of connection Objects describing edges between nodes in the composition. Each connection comprises:
- from and to** The source and destination nodes for the connection. Each of which is described by:
 - component** the identifier for the component (note that this is the local ID from among entries in the components array). If this is absent, the connection is from/to an input/output parameter for the composition.
 - param** the identifier for input/output parameter comprising the source or destination of the connection. This may be an externally-visible parameter of a specified component, or, if no component is specified then it is a parameter of the composition itself.

Here's an example of a simple composition showing the parameters (a, b, and c), components (A and B), and connections (1, 2, 3, and 4):

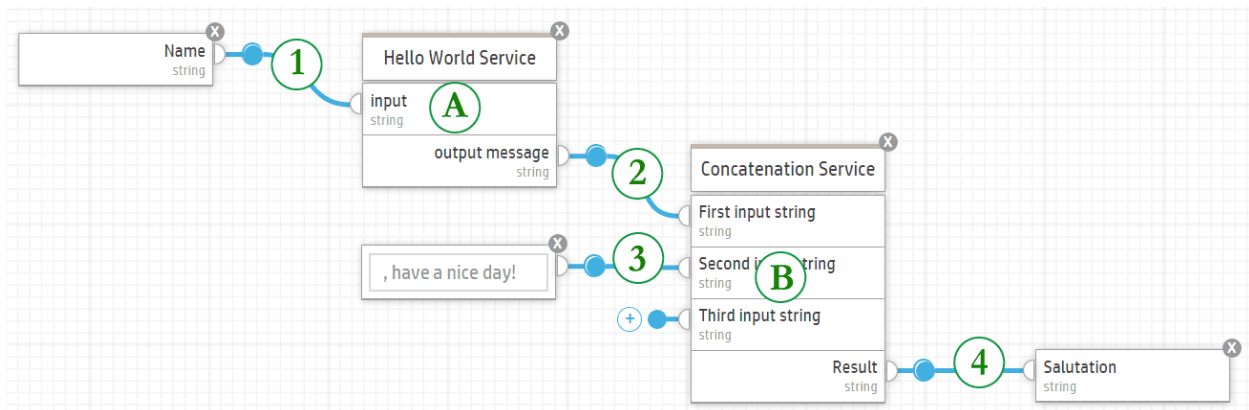


```
{
  "name":{"en":"Craig Sayers Composed Hello World Service"},
  "id":"composition_14274752025012094154699",
  "version":"1.0.1427475257095",
  "engine":"Hybrid",
  "description":{"en":"This is a very basic example service which simply says Hello."},
  "shortDescription":{"en":"Basic example service"},
  "categories":["Demonstration","Hybrid"],
  "developerName":{"en":"Craig Sayers"},
  "developerURL":"www.hp.com",
  "icon":"icon.png",
  "inputs":{
    "name":{
      "name":{"en":"Name"},
      "required":true,
      "type":"string",
      "description":{"en":"any text name"},
      "default":"CLASP User",
      "x":40,"y":40
    },
    "inputString2": {
      "name":{"en":"Second input string"},
      "description":{"en":"Second input string to be concatenated"},
      "type":"string",
      "default":", have a nice day!",
      "style":"constant",
      "x":327,"y":217
    }
  },
  "outputs":{
    "outputString":{
      "name":{"en":"Salutation"},
      "description":{"en":"Concatenation result"},
      "type":"string",
      "x":892,"y":304
    }
  }
}
```

```

"components":[
  {
    "id":"1816",
    "service":{"id":"com.hp.clasp.example.helloWorld","version":"1.0","engine":"Java"}, (A)
    "x":328,"y":40
  },
  {
    "id":"concatenator1",
    "service":{"id":"com.hp.clasp.utils.concatenator","version":"1.0","engine":"Java"}, (B)
    "x":603,"y":133
  }
],
"connectors":[
  {
    "from":{"param":"name"},
    "to":{"param":"inputName","component":"1816"} (1)
  },
  {
    "from":{"component":"1816","param":"output"},
    "to":{"component":"concatenator1","param":"inputString1"} (2)
  },
  {
    "from":{"param":"inputString2"}
    "to":{"param":"inputString2","component":"concatenator1"}, (3)
  },
  {
    "from":{"component":"concatenator1","param":"outputString"} (4)
    "to":{"param":"outputString"},
  }
],
"notice":{"en":null},
"bibliography":null
}

```



Multiple Catalogs and Sharing Services

In the current implementation every user sees every published catalog entry, and we expect users to be reasonably responsible and not deliberately break the system. That's appropriate for our target usage where catalog installations are inside a corporate firewall, and suits our common use-cases, where some services require direct access to private devices or databases. Since each organization has its own firewall, this necessitates having at least one catalog instance per organization.

Each catalog installation includes a set of built-in services. Loaded at startup, and generally standardized across installations, these are immediately available for use, providing convenient functionality out-of-the-box.

Given multiple catalog instances, it will be natural to share services among them. For manually-added services, any service developer can utilize the 'download' button beside each service on their add/edit catalog entries tab, obtaining a service.zip file which can then be dragged/dropped onto the same tab on any other catalog.

For larger numbers of services, more automation is necessary, and possible via a two-step process:

On any catalog, when an administrative user does any search they are presented with an additional "Download All" button on the results page to gather all the services matching that query (including all the service code) into a single .zip file.

The screenshot displays the CLASP web interface. At the top left is the 'hp CLASP' logo. On the top right, the user 'Hernan Laffitte' is logged in with a 'Sign out' link. A blue banner in the top right corner indicates 'ALPHA Version 0.2.12-1'. Below the header, there is a search bar containing 'cloudbit' and a 'Filter by Tag:' dropdown menu. The search results show three items: 'LittleBits CloudBit Status 1.0', 'LittleBits CloudBit Control 1.0', and 'IDOL Sentiment to CloudBit 1.0'. Each item includes a description, a language tag (Java or Hybrid), the user 'Craig Sayers', and the tags 'LittleBits, CloudBit'. A 'Download All' button is located at the bottom right of the results list.

The administrator of any other catalog can choose the sources for initial default services (those visible in the catalog which are part of the system upon startup and automatically updated whenever the system restarts). There can be a list of such sources and that list can include URLs to local or remote .zip files containing additional services.

Thus a .zip of services can be generated on one catalog, and ingested at startup time into another catalog.

Internal Service Execution Process

Elementary services

For services written in Java, the process is as follows:

1. If not already cached, then load the description.json and, using a custom ClassLoader, any necessary .jar files
2. If handling any of the inputs or outputs would require temporary filesystem storage then create a temporary task-specific working directory
3. Construct an in-memory Logger specific to this task
4. Construct the Task object, using the specified inputs and outputs.
5. Run the service using the execute(task) routine
6. During execution, the service code may get/set parameters from the Task. Some of these may be streams which requires opening/reading/writing/closing those
7. After execution completes, prepare any output parameters (which may involve persisting them and returning a URL), collect the log, and return task execution results to the caller
8. Tidy up by removing any temporary working directory and any constructed objects (including the Task and Logger).

Notice that in this process there is no state from the task remaining except any persisted results requested by the caller.

For scripted services the process is similar except that we generate a custom script, combining the requested I/O parameters with the body of the script (from the description.json) and then execute that, capturing anything written to stdout as info entries in the log, and anything written to stderr as error entries in the log.

In past implementations, we also implemented asynchronous processing, where a task is submitted with the immediate response containing not the results, but rather a link to check on the status and later retrieve results. That style is appropriate when processing takes a long time, however while simple to achieve in a quick experiment, it is tricky to handle all the corner cases correctly in the case where machines or networks may fail. In practice, we found our typical tasks executed surprisingly quickly, even for relatively large datasets, and so it was preferable to keep the implementation simple and optimize for the speed of processing tasks synchronously. In the synchronous model, an incoming request is simply directly processed by the recipient servlet and the response returned directly, thus there is no additional queue overhead, no additional transactions to check on the tasks status, and since the http connection remains open during execution, it is immediately obvious to both the caller and our server if the network or machine at either end fails.

Composed services

For composed services, the basic process for execution is to construct a dependency graph of the component parts, execute services which have no dependencies, and then remove those as dependencies from remaining services, continuing until all services have been completed. In the event any component fails, the entire execution is aborted, with the error log being returned to the user. Execution timeouts help to catch inadvertent errors.

In some special cases there is the opportunity to optimize execution. For example, if two R scripts are composed, we can generate a single script containing the body of each, where any connection between them becomes an assignment operator in the resulting script. In this way the composed service is similar in efficiency to a custom script performing the same operation as the composed service.

Our first implementation for composition was actually the most complex, where we ran each component service in a separate thread and wrapped parameters passed between services using a concurrent latch so that whenever a service requested an input parameter it would block until the preceding service had output that value. For streams we piped outputs to inputs and for sequences we used locking on the iterator, again causing the receiving service to block until the next value (or end of the sequence) was available from the preceding service. This had some desirable properties, in particular it allowed component services to run in parallel and pipelined, and we could just start all the components at once allowing the locking to control execution ordering. Error handling and logging was complex since several components could fail simultaneously, but avoiding deadlocks required additional work. Consider the following:

Developer A writes a service which generates an output stream and then writes the number of generated items to a numeric output

Developer B writes a service which reads in a numeric item count, and then reads that many items from an input stream.

Developer C combines those services connecting the output stream from A to the input stream for B, and the output count from A to the input count for B.

Now when the composed service is executed both services start, but B waits for the count before it starts to read from the input stream, while A won't write the count until it finishes writing the output, but it's limited in the ability to output since B isn't yet ready to accept the stream. Thus the composed service will timeout and Developer C may be unable to recognize the issue without internal knowledge of the component services.

Removing the deadlock is possible, but requires additional buffering, so A can write out the entire stream before B is ready to read the first byte.

That approach introduced significant complexity along with the additional overhead of thread creation, locking, and buffering even for quite simple composed services. Since we're interested in optimizing quality and throughput of the whole system rather than just speeding up any individual task, it was preferable to adopt a simpler approach running component services sequentially. While some individual services will take longer, the total work is reduced and the implementation simpler, so we can run more tasks in parallel.

For the simpler sequential implementation, we still need to consider the dependencies between services. This is achieved with a simple dependency table. For each service we list all the services from which it needs an input. Then on each pass we run all the services without any dependencies, removing them from the list of dependencies for remaining services. In the event we have services remaining and they all have at least one dependency then we must have a cycle in the composition, causing it to be immediately aborted with a suitable error message.

For passing data between component services the values of elementary variables (string, number, etc) are simply copied directly, while for output streams we need to buffer those, allowing the sending service to run before the receiving service. We found files to be both simplest to implement and surprisingly fast. Modern systems have great optimization for the case where small (less than 1Mb) files are written and then immediately read, and so we benefit from that, allowing a very simple implementation with quite adequate speed. Recall that a composed service is executed within a single servlet request on a single machine, so there is no network communication and only temporary files.

As mentioned previously, for scripted services, we may treat composed services in the same language as a special case, using judicious rewriting to simplify execution. In particular, we may combine the component scripts into a single longer script and replace parameter connections with assignment statements.

Semi-automated descriptions and schemas

We also experimented with the automatic ingestion and execution of Pig scripts². While not in our production system, it is an interesting example because the nature of Pig permitted exploration of features not easily attainable in other systems.

The core ideas were that we could both determine Relation schemas and automatically-generate human-readable descriptions from the code itself (note this is different than the other scripted languages, where we can only construct types and parameter descriptions automatically).

Example script

Here's an example Pig script which runs locally and will be a working example for subsequent sections:

```
Words = LOAD 'input' USING PigStorage() AS (word:chararray, freq:int);
ordered = ORDER Words BY freq DESC;
TopN = LIMIT ordered 500;
STORE TopN INTO 'output';
```

Schema extraction and cataloging

In common with other scripting languages we can automatically determine the input and outputs, also picking up their types and in some cases the relation schema (as in the input here), and extracting the core code:

Inputs	
Words (Words) relation (word:chararray, freq:int)	Edit
Operation	
Pig Script ordered = ORDER Words BY freq DESC; TopN = LIMIT ordered 500;	Edit
Outputs	
TopN (TopN) relation	Edit

[Submit](#)

Each schema is compared to existing catalog schemas using a similarity metric. The best case is an exact match on both types and labels, the next best is an exact match on types, and the last option is a subset of the used parameter types (again with exact names being better).

² Craig Sayers, Alkis Simitsis, Georgia Koutrika, Alejandro Guerrero Gonzalez, David Tamer Cantu, and Meichun Hsu, "The Farm: where Pig scripts are bred and raised", Demo, SIGMOD, New York, 2013.

In this case, there is a very similar schema which only differs in the name of the second parameter and so it is presented first among alternative options for the user. The idea is to encourage users to reuse existing similar schemas rather than making their own. This is also less work for the user, since the existing schema has already been documented and entered into the catalog. It also has the advantage that popular schemas will aid other users in service discovery and further encourage reuse. Should the user choose the existing schema, we automatically adjust the code, changing the parameter names to match the new schema.

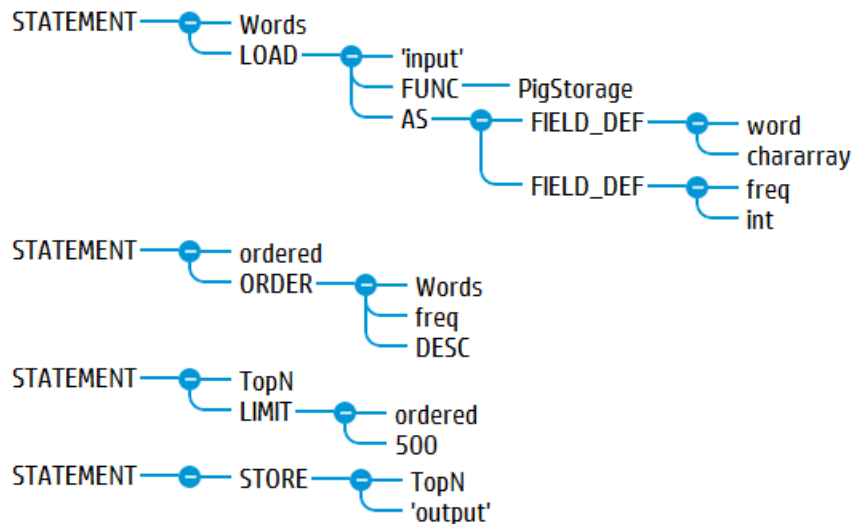
Parameter schema *
Automatically-extracted relation (word :chararray, freq :int)
Word Frequency 1 Stores a word and the number of occurrences of that word in a given text. (word :chararray, frequency :int)
N-gram Frequency 1 Stores an n-gram and the number of occurrences. (ngram :chararray, frequency :int)
Person Age 1 A simple person schema comprising name and age. (name :chararray, age :int)

If they decline to reuse a pre-existing schema, then we walk them through the process of adding/publishing as a new schema in the catalog. This is semi-automated, and just as for services, we encourage additional documentation since the user frequently possesses pertinent information unavailable in the machine-readable schema. For example, even when the schema used a readable name: 'client_birthdate', it is the user who understands that string must in ISO 8601 date format and that the client must be over the age of 18.

Generating human-readable descriptions

In addition to extracting schemas, we can create a human-readable description for the service by examining the code, building on prior work for generating natural language descriptions of SQL queries.

Here's what the code looks like in the Pig parse tree for that same Pig example script:



For each Pig expression we have an English language template which we can populate with script content.

For example, the second statement matches the template:

$((^1 \text{ ORDER } ^2 \ ^3 \text{ DESC}) \ ^4)$ → “Compute ^1 by sorting ^2 by ^3 in descending order”

So we generate for that statement:

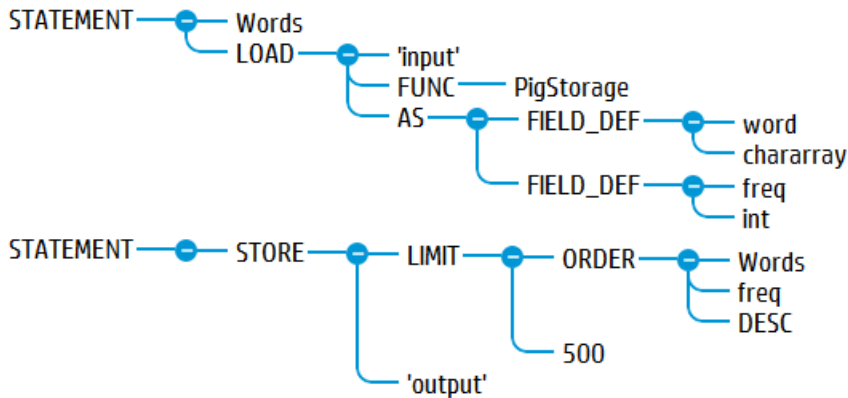
“Compute ordered by sorting Words by freq in descending order.”

In this way we can generate a description with one sentence per statement. However, the result is rather unsatisfying, both due to excessive length, and also because it uses many variable names. In this case it happens that the names are relatively meaningful, but had they just been ‘X’ or ‘Y’ the resulting description:

“Compute X by sorting Y by Z in descending order.”

is only a small improvement over the original script. Fortunately improvement is possible. The trick is to collapse statements, removing intermediate variables whenever possible – there are limits, since we don’t want sentences that are a paragraph long, but collapsing a couple of levels works well.

Using the same example as above, we can collapse to just two statements:



These can then be matched against more specialized templates, in this case the following three:

```
(STATEMENT ^1 (LOAD ^2 (FUNC ^3) (AS @4))) → "Reads in ^1 .";
(STATEMENT (STORE ^1 ^2)) → "Writes out ^1 .";
(LIMIT (ORDER ^1 ^2 DESC) ^3) → "the top ^3 ^1"
```

to produce the much more readable full text description:

Reads in Words
Writes out the top 500 Words.

and we can further summarize, generating a short description for the catalog by using the English text for the STORE statements. To produce:

Writes out the top 500 Words.

resulting in the catalog entry:



Astute readers will notice that in generating the natural language description using the more sophisticated template we left out the variable with which the top words were generated. In practice we found this and other compromises between brevity and detail acceptable, but since we have the additional information, it would be possible to enrich the description so that, for example, hovering over ‘the top 500’ could show how that was determined.

The human readable descriptions will never be as good as hand-crafted text, however they are a good starting point and do at least reflect the actual operation of the code. It is not uncommon for users to cut-and-paste

text between service descriptions, and easy to make a mistake or simply forget to change as the code evolves, and so having at least an approximate initial text helps maintain quality. Based on our own usage, we find it both less daunting and easier to modify an initial machine-generated description than to start with an empty text box and write from scratch.

Optimizing composition using schemas

Similar to other scripted languages, we can optimize service composition, creating a single script by combining the service code and using assignment operations for compositional connectors. In Pig we have the additional complexity that different services may have schemas that are not an exact match. In those cases we can perform semi-automated schema remapping. This won't always work, for example even if two schemas have a string called 'Id' with identical descriptions they may still not be the same Identifier, and so some human verification/testing will generally be necessary.

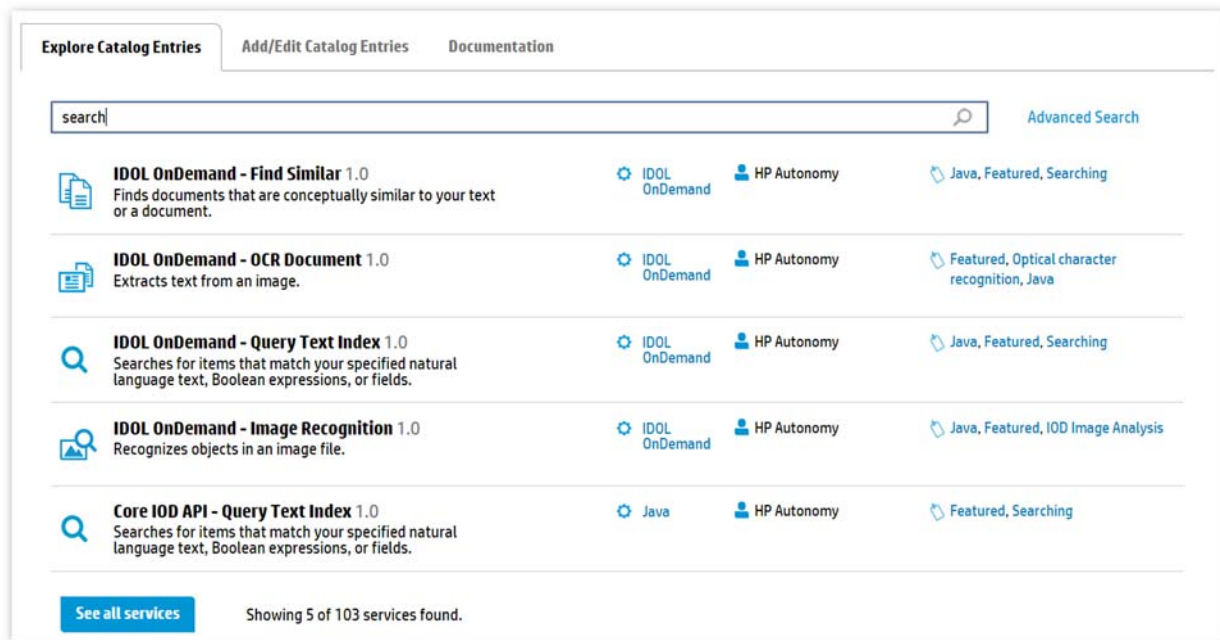
Automated text and compositions

While obviously applicable to searching for direct services in the catalog, the automated generation can also aid users in composition. For example, if a user picks one service and enters keywords for what they'd like a composition to perform, we can find compatible services, generate human-readable descriptions for the compositions and then use those both for keyword matching and to explain the results to the user.

User experience

In common with many projects, our success depends in large part on the user experience. If too hard to understand, or too cumbersome to use, then we fail. Consequently, considerable time was spent optimizing the user experience. Given the inevitable tradeoffs between perfection and expediency, and between simplicity and features, there is no perfect solution, but recent versions have proven both easier to use and more powerful.

The search interface is relatively simple, adopting the familiar text search box and leveraging the human-readable catalog text. As the number of services increased, the addition of category tags helped to organize and aid searching:



The number of category tags have themselves increased and will soon require additional assistance. In particular it will be helpful to allow search over category names in the advanced search, and to show categories as additional results for the basic text search.

It was the addition and editing of services which proved most challenging, and we went through a number of implementations and iterations. The current system aims for progressive disclosure, easing users into the process. For example, consider a new user. They start by browsing the catalog and trying out services – there is no login required for this. They just start at the home page and either click on a displayed service or search and explore. Eventually, they may experiment with composition, and at that point, with the motivation of a specific goal, we ask them to login and then take them directly to the composition interface. Advancing using the familiar 'next' button, they can either proceed all the way to publishing, or save and return later. In either case, upon leaving that page, they are shown the add/edit entries tab with the service they just created visible along with additional options for managing the service lifecycle (deprecate, delete, etc). In this way we gradually expose functionality to the user as they need it, allowing sophisticated operations without appearing too complex at any point.

Adding new coded (rather than composed) services was also challenging to simplify and we've found a drag-and-drop interface to work best there. Rather than having users choose the type of catalog entry, they just drag-drop any file and based on the type we figure out the appropriate style, prepopulating the service or dataset description. This makes the process less daunting for novice users, and, consistent with the composition editing, we use progressive disclosure, with interface affordances appearing only as they are needed. This is shown in Figure 7.

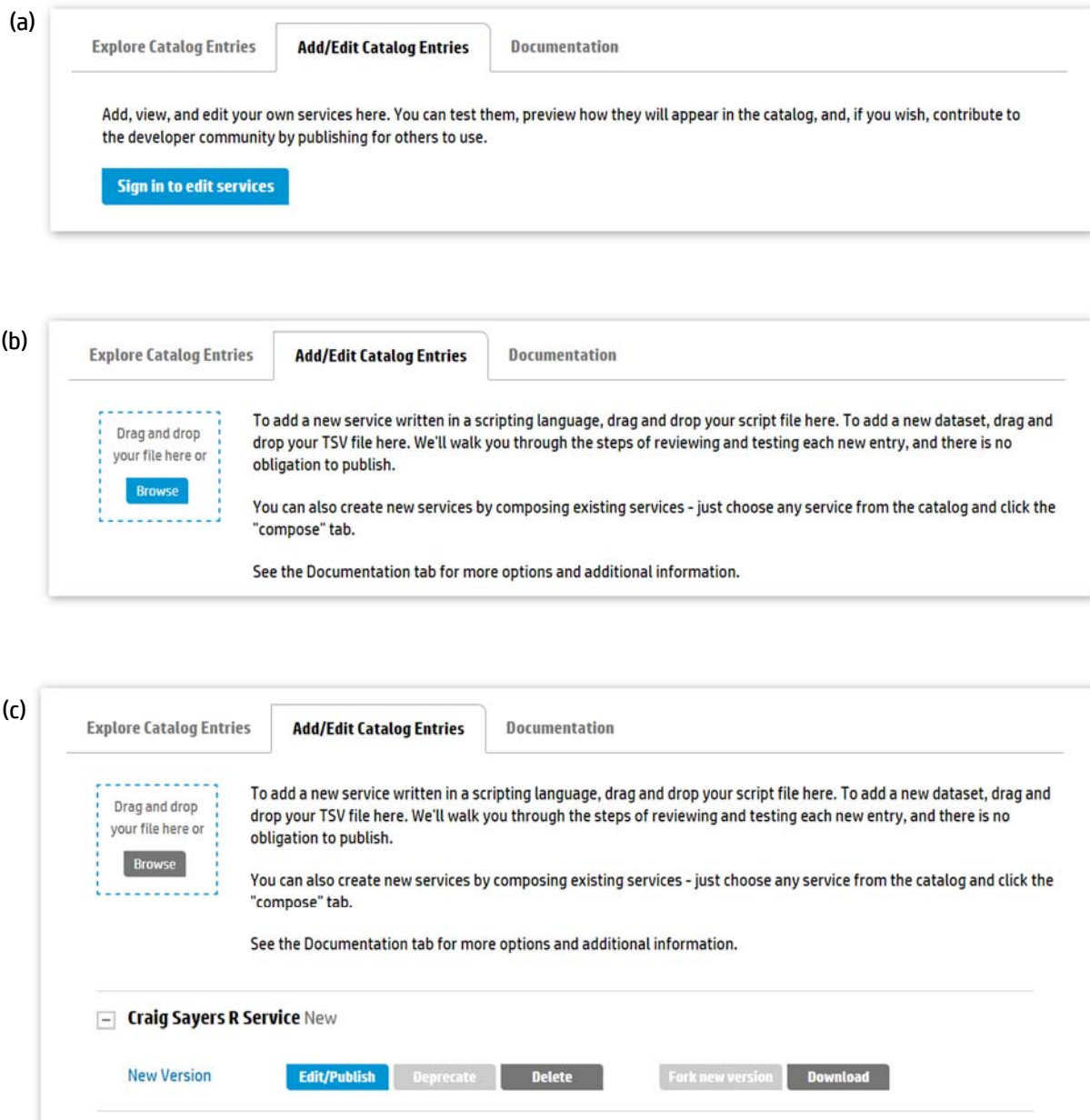


Figure 7. Progressive disclosure in the interface. On this single tab, the user sees three different views depending on their progress. Starting with signing in, then drag/drop, and then service lifecycle management.

Controls on the interface for editing and publishing services were perhaps the most challenging, and a number of iterations were necessary (see Figure 8). For example, considering just the top header of the interface: An initial tabbed interface had too much content per tab, and the progression between steps was unclear. Adding additional tabs helped simplify tab content, but adding graphics and check marks was surprisingly insufficient, with users still reporting they “didn’t know what to do next”. Replacing tabs with a more conventional progress descriptor with the familiar ‘Next’ button has proven superior.

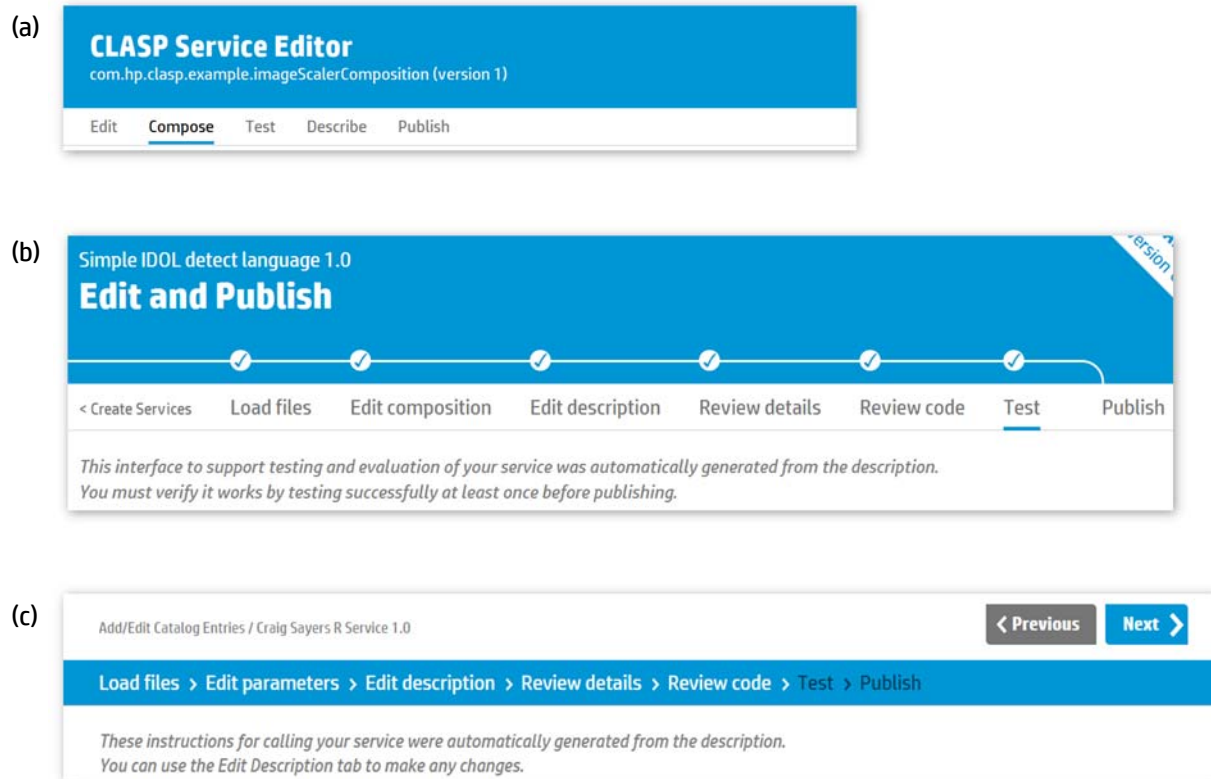


Figure 8. Edit/publish interface evolution from (a) tabs, to (b) tabs with graphic and check marks to show progress, and finally (c) a more familiar interface with next/previous buttons.

Conclusions

Our CLASP system simplifies the publication, discovery, and use of services. For service developers, it allows programmers to take a local script and turn it into a web service in just a few minutes via a simple drag/drop interface and wizard-based publication process. For application developers it provides the ability to verify service efficacy using their own data, gives example code, and provides the ability to build more sophisticated services via composition.

By simplifying the interface and using progressive disclosure we've improved the experience while adding additional features. As the user base increases, so does the number and scope of services along with our ability to make recommendations based on observations, both of which should encourage additional usage, resulting in a desirable positive feedback loop.

Currently containing more than 2,000 services and being used by more than 150 internal developers, it has shown the benefits of an app-catalog-style approach for the underlying services on which many applications depend.

Acknowledgements

The CLASP research project has been a relatively large development effort. We are particularly grateful to the developer teams in Atlantico Brazil, HP India, HP Brazil R&D, and HP GUAPO in Mexico, along with our past Summer Interns, our Labs collaborators, local Research Engineering Services, our business unit collaborators across the company, and our management for their ongoing support.