Last modified: July 8, 2014

Getting started with Bisque

This document is intended for users who would like to integrate their analysis code into the Bisque system. Assuming that you already have that code and the input required to run it (files and parameters), this overview outlines steps needed to get started with Bisque [1].

1 Preliminary steps

1.1 iPlant account

Use Trellis [5] to create an account and log-in to the dashboard. One of the services listed under "Available services" is Bisque.

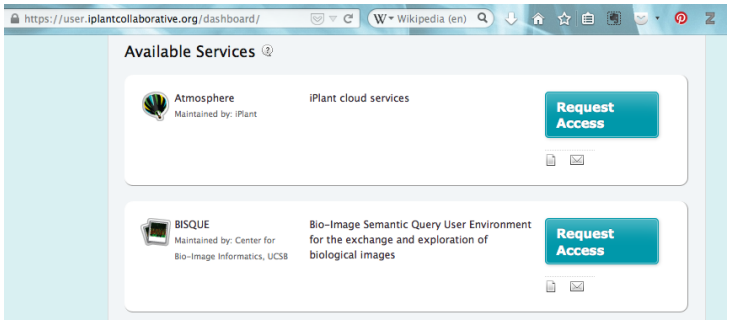

Figure 1: The services available through your Trellis dashboard.

Once the request is approved, use the "Go to BISQUE" [2] link to access the Bisque database hosted by iPlant (see Fig.1.1).

1.2 Analysis code

Select an existing program to turn into a Bisque module. This analysis code can be written in any language, and with the help of the Bisque API, it can be "wrapped" and connected to the Bisque services.

For the purposes of this tutorial, I am using a C++ program that finds points of interest in images. I created an executable binary, which I can run on the command line.

```
./find_points --output-directory=$OUTDIR --has-tip-growth --min-blob-size=5
--max-blob-size=50 --num-threads=8 $INDAT/Pos001_S001_t%02d_z%02d_ch00.tif
```

The above command shows the parameters that are required to run the code: `$OUTDIR` is a path to an output directory, `$INDAT` is a path to where the input files are, with the input file names described using a Unix printf-style pattern.

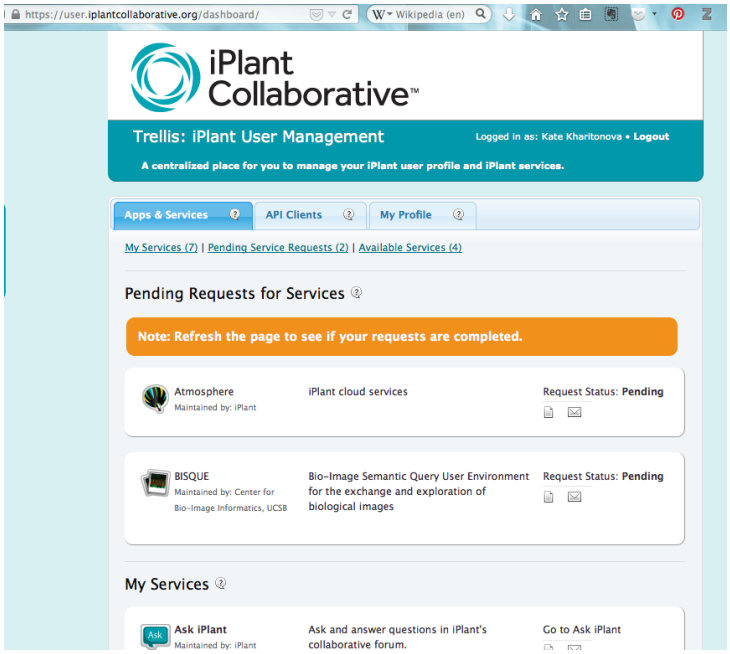

Figure 2: Pending requests.

1.3 Overview of steps

Before we dive in, let's look at the broad picture of the work ahead of us. Following a Bisque module creation tutorial [3], below are the steps for setting up a working module:

1. Creating a module definition XML document
2. Making the module code work with Bisque (either by modifying the module code to use the Bisque API or by writing a wrapper / conversion glue code)
3. Creating a web server or running the Engine Server

   Engine Service - a configuration file on how the Engine Service will run your code
4. Registering the module with Bisque

2 Creating a module definition XML document

This section is based on the instructions listed on the Bisque Module Specification page [4]. After providing an overview of the module system, I outline its various components, constructing an example that uses my analysis code.

2.1 Module system overview

The Bisque Module System aims to provide services to the user-created analysis code, so-called analysis module. Every module, before it is used, must be declared to Bisque through module registration.

To be used with the Bisque system each module must define a module-definition file, which allows the module to be registered. Registration involves posting the module definition file to a module manager, which can then dispatch execution requests back to the module. The EngineScripts wraps local scripts and creates a web-service based on module definition file. The EngineServer (installed with Bisque) will manage registration and execution of the local script. It will issue an execution request (MexRequestSpecification), which contains parameters needed to execute the module on behalf of a user.

2.2 Module definition file

The module-definition file is an XML-encoded file that provides attributes associated with the module. Each module must have a unique name and formally specify its input and output parameters. In addition to input and output, you can specify the coding language, authorship, and code version.

The module definition document is actually a templated MEX (Module EXecution) document. The template parameters, for example, can be used to render the user interface (UI) for a module if the programmer does not want to fully implement the UI. If, however, the programmer would like to provide a fully customized user interface then the input parameters can be configured to do so. But, let's leave customization for later: it is easiest to start with the automatically provided UI and then customize it if the need arises.

Every module definition file, at a minimum, should specify the following definitions

- inputs
- outputs
- title
- description
- authors
- thumbnail

This means that we can create a bare-bones template (where words in ALL-CAPS are meant to be replaced).

```
<module name="MY_MODULE" type="runtime">
    <tag name="inputs">
    </tag>

    <tag name="outputs">
    </tag>

    <tag name="title" value="TITLE" />
    <tag name="description" value="This module DESCRIPTION" />
    <tag name="authors" value="AUTHORS" />
    <tag name="thumbnail" type="file" value="public/thumbnail.png" />
</module>
```

2.3 `inputs` tag

All input needed by the module has to be declared inside the `<tag name="inputs">` block (i.e. between the `<tag ...>` and `</tag>`). The input can be of the following types:

- `system-input`: element required by the module and filled by the system
- `formal-input`: element required by the module and filled by the UI

2.3.1 `system-input` type

Below is the list of names that are used with the `system-input` type ((!) indicates a required name):

- `mex_url` (!): a URL pointing to the MEX document; necessary for the module to do anything meaningful
- `bisque_token` (!): a token that is used for authentication, without it module can't write anything
- `module_url`: a URL to the module

Now we can add the following required inputs to our module definition file:

```
...
    <tag name="inputs">
        <tag name="mex_url"      type="system-input" />
        <tag name="bisque_token" type="system-input" />
    </tag>
...
```

2.3.2 `formal-input` type

The formal inputs may be of two ways:

- a link to an existing resource
- a complete resource in-place

Let's look at them closer.

Existing resource

A link to an existing resource is declared as a tag with a type of a resource you would like to point to. For example, to link to an image

```
<tag name="image_url" type="image" />
```

A link to a resource will get send to the module inside the MEX document looking like this:

```
<tag name="image_url" type="image" value="http://host/path" />
```

The name here is a user-defined name and will only be used by the module to find the right resource. The type can be any resource type existing in the system. For example you can use: image, dataset or tag. Automated interface will try to generate an appropriate resource picker, for example, a browser for an image or a dataset-selection dialog box for a dataset.

A resource in-place

Any resource and tag with a non-resource (user-given) type is a resource in-place. They are useful to pass information that should not be stored in the system, such as a numeric parameter or a graphical annotation.

```
<tag name="radius" type="number" />
```

Continuing with our analysis code example, we see that our binary takes several input parameters.

```
--has-tip-growth
--min-blob-size=5
--max-blob-size=50
--num-threads=8
$INDAT/Pos001_S001_t%02d_z%02d_ch00.tif
```

As was described above, our numeric parameters are the "resource in-place" (the boolean parameter `--has-tip-growth` can easily be made numeric by using values 1 and 0), and the image input directory is the "existing resource". Now we can add them (and their default values) to the inputs tag in our module definition file:

```
...
    <tag name="inputs">
        ...
        <tag name="has-tip-growth" value="1" type="number" />
        <tag name="min-blob-size" value="5" type="number" />
        <tag name="max-blob-size" value="50" type="number" />
        <tag name="num-threads" value="8" type="number" />

        <tag name="image_url" type="image" />
    </tag>
...
```

References

[1] Bisque database. http://www.bioimage.ucsb.edu/bisque. 1

[2] Bisque database. http://bisque.iplantcollaborative.org. 1

[3] Bisque module creation tutorial. http://biodev.ece.ucsb.edu/projects/bisquik/wiki/Developer/ModuleSystem/Tutorial. 2

[4] Bisque module specification. http://biodev.ece.ucsb.edu/projects/bisquik/wiki/Developer/ModuleSystem/BisqueModuleSpecification. 2

[5] Trellis: iplant user management. https://user.iplantcollaborative.org. 1