Last modified: July 15, 2014

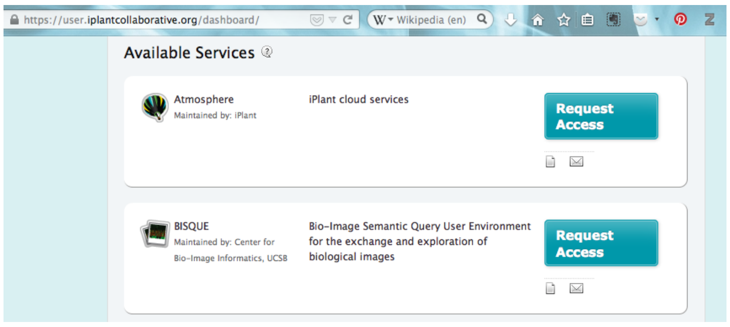
Getting started with Bisque

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].

0 Preliminary steps

0.1 iPlant account

Use Trellis [6] to create an account and log-in to the dashboard. One of the services listed under "Available services" is Bisque.



Available Services

Atmosphere
Maintained by: iPlant
iPlant cloud services

BISQUE
Maintained by: Center for Bio-Image Informatics, UCSB
Bio-Image Semantic Query User Environment for the exchange and exploration of biological images

Figure 1: The services available through your Trellis dashboard.

Once the request is approved, use the "Go to BISQUE" [3] link to access the Bisque database hosted by iPlant (see Fig.0.1).

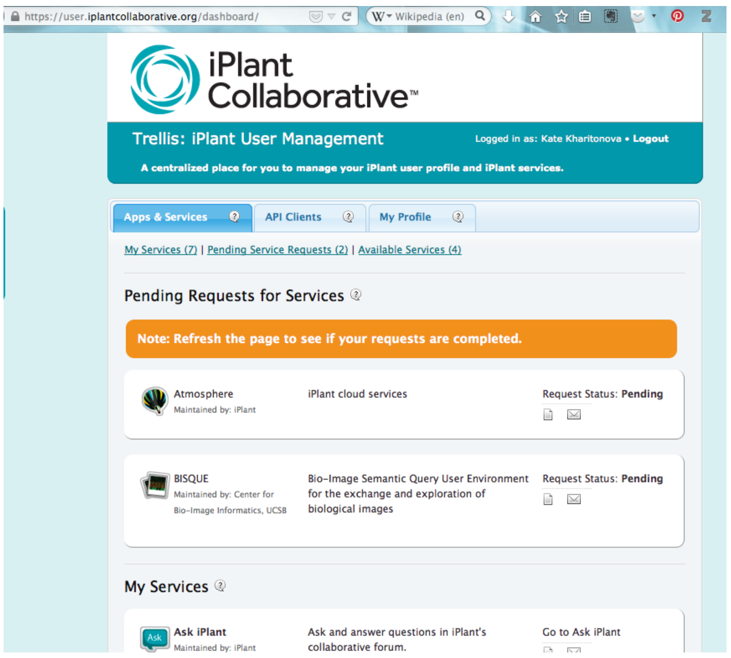
0.2 Analysis code

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 (this is what this tutorial is about).

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, where `$OUTDIR` is a variable that holds the path to an output directory, `$INDAT` is a path to where the input files are, and `Pos001_S001_t%02d_z%02d_ch00` is the Unix printf-style pattern, which describes the format of the input file names.


https://user.iplantcollaborative.org/dashboard/

iPlant Collaborative™

Trellis: iPlant User Management

Logged in as: Kate Kharitonova • Logout

A centralized place for you to manage your iPlant user profile and iPlant services.

Apps & Services | API Clients | My Profile

My Services (7) | Pending Service Requests (2) | Available Services (4)

Pending Requests for Services

Note: Refresh the page to see if your requests are completed.

Atmosphere Maintained by: iPlant	iPlant cloud services	Request Status: Pending
BISQUE Maintained by: Center for Bio-Image Informatics, UCSB	Bio-Image Semantic Query User Environment for the exchange and exploration of biological images	Request Status: Pending

My Services

Ask iPlant Maintained by: iPlant	Ask and answer questions in iPlant's collaborative forum.	Go to Ask iPlant

Figure 2: Pending requests.

0.3 Overview of steps

Before we dive in, let's look at a broad picture of the work ahead of us. Don't worry if these steps don't yet make a lot of sense: I will describe each step in more detail in the subsequent sections.

The general idea is this: you have your code that you would like to run using the Bisque system. To do that, you need to describe your module/analysis code to Bisque, i.e. tell Bisque the values for your code's parameters as well as the location/type of the input files. This is taken care of by the module definition XML document. The module definition file describes a web interface that the system will present to you, so that you can set these parameters. You will also need to specify the format/structure and types of the output files in that XML file.

The next step is to make sure that your code can talk to the Bisque system. This is accomplished by using the Bisque API [2]. This API allows you to write what's called a "wrapper": a layer of conversion code between your analysis/module code and the Bisque system. It glues together the pieces needed by both layers. Some of its many functions include processing the module definition file, setting up necessary directories, converting data from the Bisque format into the input format expected by your analysis code, running your analysis code and converting the outputs.

In order to properly test how well your analysis works with Bisque, you need to have access to the Engine Server. The Engine Server is what allows Bisque to talk to many modules that may exist on many separate machines. A Bisque package called the Engine Service allows the developers to wrap local files, give them URLs and provide the necessary entry points to facilitate communication between the modules and Bisque. Think of it as a translator that converts text from one language, so that a person who speaks a different language can understand it.

Finally, in order to allow the main Bisque server to access your module, you need to register the Engine Server that's running your module. The registration makes the main Bisque server aware of the existence of your server, and allows them to "talk" to each other and exchange files.

In summary, below are the steps for setting up a working module taken from the Bisque module creation tutorial [4]:

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 (i.e. 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

Let's take a look at each of the steps in more detail.

1 Creating a module definition XML document

This section is based on the instructions listed on the Bisque Module Specification page [5]. After a brief overview of the module system, we'll outline its various components, and construct an example that uses our analysis code.

1.1 Module system overview

Initially, all you have is your analysis code, also known as the analysis module, and you would like to use the Bisque Module System. Every module, before it is used, must be declared to Bisque through module registration. In order to register your module, you must define a module-definition file written in XML.

Through the registration, the module definition file is posted to a module manager, which can then dispatch execution requests back to the module. Don't worry if this doesn't make much sense right now – an understanding of this process is not required to create a module definition file. Just know that once you create this XML file and register your module, the Engine Server will be using it to wrap the local scripts and create a web-service based the provided module definitions.

1.2 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

Let's create a bare-bones template with the required tags, 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>
```

1.3 `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

1.3.1 `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>
...
```

1.3.2 `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>
...
```

Side note: I chose to keep the names of the tags the same as the analysis code arguments. This is not required, as is just a matter of convenience (I could have just named them param1, param2 and left it to the wrapper to match them to the correct arguments).

2 Making the module code work with Bisque

References

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

[2] Bisque api. http://biodev.ece.ucsb.edu/projects/bisquik/wiki/Developer/BisqueApi. 2

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

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

[5] Bisque module specification. http://biodev.ece.ucsb.edu/projects/bisquik/wiki/Developer/ModuleSystem/BisqueModuleSpecification. 3

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