Getting started with Bisque

Yekaterina Kharitonova and Andrew Predoehl

September 29, 2014

Contents

Getting started with Bisque

This document is about the specifics of using Bisque to run and share an analysis module. It extends and builds on the material of a related document, "Bisque module integration," which serves as a high-level overview of the Bisque system, from the perspective of a module developer. In contrast, this tutorial 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 [9] to create an account and log-in to the dashboard. One of the services listed under "Available services" is Bisque.

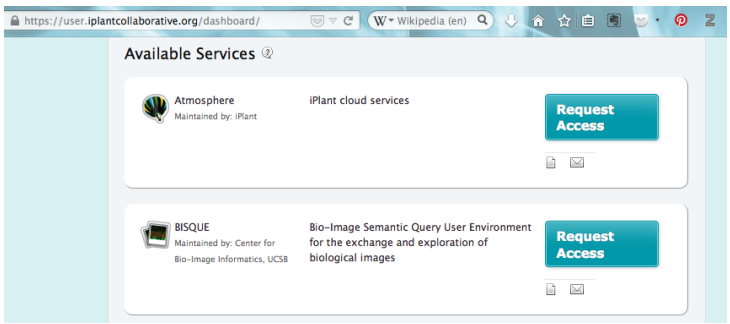

Figure 1: The services available through your Trellis dashboard.

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

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.

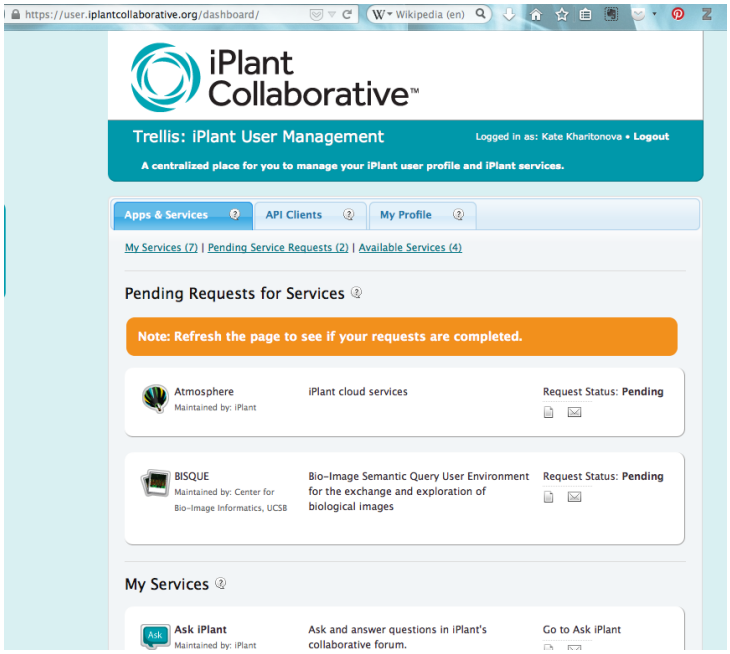

Figure 2: Pending requests.

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 would like to run some code 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 [7]:

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 [8]. 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. Before it is used, every module 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 on 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 optionally specify the coding language, authorship, and code version.

The module definition document is actually a templated MEX (Module EXecution) document [6]. 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. Since writing a customized interface requires knowledge of HTML, CSS, etc., it is beyond the scope of this tutorial. If you are interested, you can find an example of a module definition file on the Bisque Module specification page [8].

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 *like this* are meant to be replaced.

```
<module name="module-name" type="runtime">
<tag name="inputs">
</tag>
<tag name="outputs">
</tag>
<tag name="title" value="module-title" />
<tag name="description" value="module-description" />
<tag name="authors" value="authors" />
<tag name="thumbnail" type="file" value="path-to-image-file" />
</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. An exclamation mark, (!), after the type name 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 other type of `system-input` type is `formal-input` type. The formal inputs may be of two ways:

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

Let's take a closer look at them.

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" />
```

The "name" attribute is selected by the user, and the module will be using your selected name to find the right resource. The "type" can be any resource type that is recognized by the Bisque system. For example, you can use: image, dataset or tag. Automated interface will try to generate an appropriate resource picker depending on the provided type, 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 analysis code 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 flag `--has-tip-growth` can easily be made into a boolean parameter 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 "name" tags the same as the analysis code arguments. This is not required, as it 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

In order for Bisque to be able to run your code, it needs to know how to interpret and handle the input parameters, and also how to process the output results to display them on the Bisque page. There are many ways to do it, such as using Matlab, command line and Python APIs (see the examples on the Wiki tutorial page [7]).

In this document, I will be using Python to illustrate how to write a simple module "wrapper", which will let you test your interaction with Bisque. The main idea of this section is to help you build the scaffolding necessary to get started with the module execution and see how to use some of the Bisque API functions.

2.1 Creating a "wrapper"

How do you start? To begin with, let's create a skeleton Python file (`get_points.py`) that will import the necessary libraries to allow us to do simple event logging and argument processing. The comments in the code will provide more information.

Listing 1: `get_points.py` beginning.

```
1 #!/bin/env python
2 #
3 # Language:  Python 2.x
4 #
5 import os
6 import sys
```

```
7  import logging
8  import glob
9  import re
10 import bq.api
11 import gobject
12
13 # The name of one of the child elements of the 'input' tag; also the name
14 # of one of the script arguments (consequently). The tag value contains the
15 # URL of the main input resource, i.e., the 4D image the user wishes to
16 # analyze.
17 img_url_tag = 'image_url'
18
19 # Value is (eventually) the pathname of the MEX staging directory, which
20 # contains the executable(s), and the subdirectories holding input & output.
21 staging_path = None
22
23 # Dictionary that stores key/value pairs given to this script at invocation
24 # by Bisque, as command-line arguments.
25 named_args = {}
26
27 if __name__ == '__main__':
28     # use regular expressions in order to get the base name
29     # of the file executing this cide and use it as the log file name
30     self_name = re.match(r'(.*)\.py$', sys.argv[0]).group(1)
31
32     # start some logging (DEBUG is verbose, WARNING is not)
33     log_fn = self_name + '.log'
34     logging.basicConfig(filename=log_fn, level=logging.WARNING)
35     #logging.basicConfig(filename=log_fn, level=logging.DEBUG)
36     logging.debug('Script invocation: ' + str(sys.argv))
37
38     # read key=value pairs from the command line
39     #-------------------------------------------------------------------------
40     for arg in sys.argv[1:]:
41         tag, sep, val = arg.partition('=')
42         if sep != '=':
43             error_msg = 'malformed argument ' + arg
44             logging.error(error_msg)
45             raise Exception(error_msg)
46         named_args[tag] = val
47         logging.debug('parsed a named arg ' + str(tag) + '=' + str(val))
48
49     # Three mandatory key=value pairs on command line
50     #-------------------------------------------------------------------------
51     murl = 'mex_url'
52     btoken = 'bisque_token'
```

```
    for required_arg in [btoken, murl, img_url_tag]:
        if required_arg not in named_args:
            error_msg = 'missing mandatory argument ' + required_arg
            logging.error(error_msg)
            raise Exception(error_msg)
```

As you will see in the next section, we are running a Bisque session in a "Staged" environment. This means that the execution looks inside a temporary directory for the necessary executable files. The path to this directory is provided as one of the input parameters.

Listing 2: Set up a staging directory

```
    # Record staging path (maybe on cmd line)
    #----------------------------------------------------------------------
    stage_tag = 'staging_path'
    if stage_tag in named_args:
        staging_path = named_args[stage_tag]
        del named_args[stage_tag] # it's saved, so delete it from named_args
    else:
        staging_path = os.getcwd() # get current working directory
```

We are now ready to pass the values of `mex_url` and `auth_token` to an API function `init_mex()` and start the Bisque session.

Listing 3: Start a Bisque session.

```
    # establish the connection to the Bisque session
    #----------------------------------------------------------------------
    logging.debug('init bqsession, mex_url=' + str(named_args[murl])
                          + ' and auth_token=' + str(named_args[btoken]))
    # Starting a Bisque session
    bqsession = bq.api.BQSession().init_mex(named_args[murl], named_args[btoken])
    del named_args[murl] # no longer needed
    del named_args[btoken] # no longer needed
```

Once the session is started, let's run a simple test that gets the input images' path from Bisque and outputs the command assembled using the provided parameters.

Listing 4: Running a test function.

```
    try:
        outputs = test_find_points(bqsession, named_args[img_url_tag)
    except Exception as e:
        logging.exception(str(e))
        bqsession.fail_mex(str(e))
    else:
        bqsession.finish_mex(tags=[outputs])
```

Before we run the `test_find_points()` we need to assemble together the named arguments that were provided to Bisque. We use a short function, which would look up

the argument names in the `named_args` dictionary and style them as `--argument=value`. This function will be called from another function, which would assemble these strings all into a single command.

Listing 5: Building the list of arguments and their values.

```
def build_these_opts(option_names):
    '''
    Given a list of names of fields in the input tag, we construct a
    list of those field names and values in the style of GNU long
    options, using the same field name.
    '''
    return ['--' + x + '=' + str(named_args[x]) for x in named_args]

def build_incantation():
    logging.debug('assembling the incantation')

    # start building this incantation with program name and
    # some default arguments
    incantation = [os.path.join(staging_path, 'find_points')]
    incantation.append('--has-tip-growth')
    incantation.append('--num-threads=8')

    # append options for points detection
    incantation.extend(build_these_opts(['min-blob-size', 'max-blob-size']))

    # Return the assembled result
    return [incantation]
```

And, finally, we add the `test_find_points()` routine.

Listing 6: An example `test_find_points()` function.

```
def test_find_points(bqs):
    '''
    Python wrapper:  This is the high-level function that
    invokes all the lower-level programs given that a Bisque session has been
    started.  Errors must be handled by raising exceptions.

    bqs is a Bisque session object (part of the Bisque API).
    '''
    # update the interface button and invoke the function
    #-----------------------------------------------------------------------
    bqs.update_mex('testing incantation')
    incantation = build_incantation()
    logging.debug('assembled incantation: ' + incantation)
```

2.2 Describing execution

Now that we have a "wrapper" code, it is necessary to describe to Engine Service how to run it. This is done using two files: `setup.py` and `runtime-module.cfg`.

2.2.1 `setup.py`

`setup.py` is responsible for preparing the code for execution. For example, if your code needs to be compiled, this is the file where you would include the compilation steps. Here you can also add the dependency check for the existence of required libraries or executables.

The following are the contents of the `setup.py` file for our test program.

Listing 7: Contents of the `setup.py` file

```
# Install script that runs 'find_points'
import sys
from bq.setup.module_setup import python_setup, require, read_config

def setup(params, *args, **kw):
    python_setup('get_points.py',  params=params )

if __name__ =='__main__':
    params = read_config('runtime-bisque.cfg')
    if len(sys.argv)>1:
        params = eval(sys.argv[1])
    sys.exit(setup(params))
```

2.2.2 `runtime-module.cfg`

This config file describes the module's code files and how to run them. Below are the main parameters that you can configure in this file:

files

Lists the files (scripts, binaries), necessary for the execution.

runtime.platforms

Currently supported platforms:

- condor
- command

The order in which they are listed in the config file defines preference in usage. E.g.

```
runtime.platforms = condor, command
```

environments

Currently supported environments:

- **Staged**: Creates a temporary directory where all executable files are placed for running (in condor it would be a shared directory). Same directory is used for storing temporary files.
- **Matlab**: Sets necessary environment variables to run Matlab code or compiled Matlab scripts.
- **MatlabDebug**: Sets necessary environment variables to run Matlab and runs interpreter with the desired function.

The following are the contents of the `runtime-module.cfg` file for our test program.

Listing 8: Contents of the `runtime-module.cfg` file.

```
# Module configuration file for local execution of modules
module_enabled = True
runtime.platforms = command

[command]
executable  = python get_points.py
environments = Staged
files = get_points.py, gobject.py, metadata.py, find_points
```

3 Engine server

3.1 Overview of the engine server

According to the Bisque developers' wiki [4], "The engine server is actually not strictly part of the Bisque infrastructure, but is used to create web-accessible analysis from simple scripts that Bisque can use. Engine servers are used to wrap C++, matlab, python, modules to create a URL for each module entry point that MEX requests can be dispatched to.

Engine severs can be deployed remotely allowing users to develop modules that interact with Bisque system elsewhere."

In short, after you create your module definition file (as described in Section 1), you can start your engine server (or use one that's provided to you). The server will be able to locate your modules, "and register them within the bisque server of choice." [4]

3.2 Setting up an engine server instance

This section assumes you have basic familiarity with a Unix-style command-line environment, such as how to use commands like `cd`, `mkdir`, `cp`, and others. If this is totally new for you, and you have enough time, it might be a good idea first to read a basic tutorial on the Unix-command line environment, also known as the "bash shell." However, if you are in a hurry, you can just follow the steps below. They will work even if you aren't sure

what they are doing, but pay close attention to the spacing, and notice that some of the steps are fill-in-the-blank.

This tutorial adapts the 'engine' installation instructions found on the Bisque Installation page [5]. We assume you have obtained an iPlant account hosted on machine `rogers.iplantcollaborative.org`, and now you are setting up an engine on this machine, because it already has all the mandatory background software in place.

1. Choose a port number for your engine.

   Since 'rogers' is behind a firewall, your engine must communicate on a TCP port[1] between 11000 and 12000. Just pick a random port number in that range, and test whether it is available using a web browser on any machine with an internet connection. For example, suppose you want to test whether port number 11000 is available. You would point a web browser to URL

   `http://rogers.iplantcollaborative.org:11000/engine_service`

   and see if you get a reply. If there is already an engine server using that port (as in this example) your browser should receive an HTML page listing the modules currently offered by that engine. But if the browser instead says "failure to connect" or something similar, the port number is probably available for you. (This method is not foolproof, but we will cope with that later.) You might need to try a few guesses to find an unused port number. Test each guess by trying to browse the URL

   `http://rogers.iplantcollaborative.org:`*`portnumber`*`/engine_service`

   ... where you fill in the *`portnumber`* blank with your guess.

2. Log in to 'rogers'.

   You can access rogers using the *ssh* command from an internet-connected Linux or Mac OS X machine, typing commands into the terminal. Type the command below, substituting the username of your account on rogers in the *`username`* blank.

   `ssh -p 1657 `*`username`*`@rogers.iplantcollaborative.org`

   From a Windows computer, you might instead want to use the SSH terminal program *putty* to access rogers, via its SSHD port 1657. Please see the putty documentation for more details.

   Email support@iplantcollaborative.org if you run into any problems.

3. Create a top level `bisque` directory and download the setup Bootstrap script.

   ```
   mkdir bisque
   cd bisque
   wget http://biodev.ece.ucsb.edu/binaries/depot/bisque-bootstrap.py
   ```

[1] Maybe you are asking, "What is a TCP port?" Answer: it is part of an internet address. In this case, it is part of the address of your engine server. Our machine's full DNS name, `rogers.iplantcollaborative.org`, is like the address of an apartment building, where many engine servers "live" in separate apartments. The port numbers are like apartment numbers: each engine has its own port.

4. Create a Python virtual environment and install all python dependencies for Bisque. (Python 2.6 or 2.7 is required).

   ```
   python bisque-bootstrap.py
   ```

5. Activate the Bisque virtual environment.

   ```
   source bqenv/bin/activate
   ```

6. Automatically configure the Bisque engine.

   Most of the configuration is done by the two commands below. You will be asked a series of question about your intended installation. You can read more about this step on the Bisque Installation page [5].

   ```
   paver setup engine
   bq-admin setup engine
   ```

7. Configure the port number of the engine

   The file `config/site.cfg` needs some extra changes. We must replace the default port number of 27000 with your own port number chosen in step 1. If you already know how to edit text files like this, you can do so using a text editor like Emacs or Vi. Otherwise, you can use the 'sed' command below, filling in the ***portnumber*** blank with your port number.

   ```
   URL=rogers.iplantcollaborative.org
   sed -i.backup -e s/$URL:27000/$URL:portnumber/ config/site.cfg
   ```

   (This modifies the `config/site.cfg` file, and retains the former contents in a file named `config/site.cfg.backup` as a safety measure. Thus, if anything goes wrong, you could undo the 'sed' command by entering

   ```
   mv config/site.cfg.backup config/site.cfg
   ```

   . . . but do not do this unless it is necessary.)

8. Start the system.

   ```
   bq-admin server start
   ```

9. Test the system.

   We will do two final tests to see if the system is working now. First, enter the commands below and examine the system's response to each one.

   ```
   BPN=bisque_portnumber && echo "No problem"
   ls $BPN.pid && echo "No problem"
   ls $BPN.log && echo "No problem"
   grep "already in use" $BPN.log || echo "No problem"
   ```

   If each command generates output that includes the line "No problem," then your system passes the first test. Otherwise, the server was not able to start.

Address already in use? Specifically, if the last command outputs something like "socket.error . . . Address already in use," then the port number you chose in step 1 might not have really been available after all. (Possibly another engine is actually occupying the port number you picked, and it was just temporarily unresponsive when you tried to load its URL.) The steps below will help you recover from this problem.

- Enter `bq-admin server stop` and ignore any error messages.
- Repeat step 1 above to find a new port number.
- Perform the "mv" command described in step 7 to undo the previous "sed" command (if you had used sed in step 7).
- Repeat steps 7, 8, and 9 using the new port number.

The second test is to point a browser (on an internet-connected machine other than rogers) to your engine's URL,

`http://rogers.iplantcollaborative.org:`*`portnumber`*`/engine_service`

. . . substituting your own port number in the blank. If the browser shows a page listing the modules on your server, then congratulations, you are now running a Bisque engine server!

If either test fails, you might find more diagnostic information in the log file `bisque_`*`portnumber`*`.log`.

10. Exit the shell and disconnect from rogers:

    `exit`

3.3 Modifying your engine server

Usually, the instructions in section 3.2 need to be performed just once. However, any time you add or delete a module, modify your module's name, change its module-definition file, change the engine configuration, or alter other aspect of the engine that affects its interaction with Bisque, then you must restart your engine server. Bear in mind that if you restart the engine while one or more users are using your modules, their work will be lost.

1. Log in to rogers (§3.2, step 2).
2. Enter the bisque directory.

   `cd bisque`

3. Activate the Bisque virtual environment (§3.2, step 5).

   `source bqenv/bin/activate`

4. (Optional.) Test whether anyone is currently using your engine.

   TBD.

5. Restart your engine.

   `bq-admin server restart`

4 Registering the module with Bisque

In this section, we assume that you have set up a module in either a full Bisque server or an engine server, but we do not assume your server is necessarily hosted on machine `rogers.iplantcollaborative.org`, as we did in sec. 3.

When you have your module ready and would like to deploy it, you need to make Bisque aware of your module's existence. To do this, you first need to restart the Bisque service by issuing the following commands, within the Bisque virtual environment (see §3.2, step 5):

```
bq-admin server restart
```

If you followed section section 3.2, and set up an engine on host 'rogers,' then you also chose a TCP port number for your engine. This means, that in order to see your module in the engine's list of available modules, you can use the following URL from the browser of any internet-connected computer:

`http://rogers.iplantcollaborative:`*`portnumber`*`/engine_service`

... provided you substitute your chosen port number in the blank. If you see your module on the list, it means that it was successfully picked up by the engine service and is available for registration with your Bisque site. Similarly, if you configured an engine elsewhere, you can check a similar HTML page to see if your module is visible.

`http://`*`host`*`:`*`portnumber`*`/engine_service`

... where *`host`* is either the DNS name of your server (full Bisque server or engine server), or it is `localhost` if you are only accessing this server locally. The appropriate value of *`portnumber`* is found in your server's `site.cfg` configuration file, although its default value is 27000.

If you are configuring a complete Bisque server, be aware that the default Bisque configuration will only look for new modules on `localhost`, so the only modules to be registered with the Bisque server are those running *locally* (i.e., on the same machine as your Bisque server.) If you would like your Bisque server to register a module on a remote engine, currently it is done using the admin interface on the Bisque server (use `-p register` if you'd like to make your module public):

```
bq-admin module -u username:password register \
 http://engine-host:portnumber/engine_service/module-name
```

On the other hand, if you want the iPlant Bisque server, publicly available at domain `bovary.iplantcollaborative.org` to register and present one of your modules on a remote engine server (possibly on 'rogers'), send email to support@iplantcollaborative.org with your request, and it will be handled manually.

Congratulations! You should now be able to access your module at the main URL of

your Bisque install:

`http://`*`bisque-host`*`/module_service/`*`module-name`*

. . . where *`bisque-host`* is either your own Bisque server, or iPlant's public Bisque server, `bovary.iplantcollaborative.org`.

If you are unable to access your module or have encountered any other problems, you can address your questions to the `bisque-bioimage google group` (`https://groups.google.com/group/bisque-bioimage`).

Acknowledgements and meta-text

Much of this work was derived from the Bisque Developer Wiki [7]. I recommend using it if you'd like more information and instructions (e.g., the specifics of the Bisque API [2]).

The authors wish to thank Kris Kvilekval, Dmitry Fedorov, Martha Narro, and Sabrina Nusrat for their assistance, which was made possible in part through iPlant's Extended Collaborative Support program.

This document was last modified on September 29, 2014. Its version control identity string is below:

```
$Id: bisque.tex 17510 2014-09-09 23:47:36Z predoehl $
```

References

[1] About bisque database. `http://www.bioimage.ucsb.edu/bisque`. 2

[2] Bisque api. `http://biodev.ece.ucsb.edu/projects/bisquik/wiki/Developer/BisqueApi`. 3, 17

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

[4] Bisque engine server. `http://biodev.ece.ucsb.edu/projects/bisquik/wiki/Developer/ModuleSystem/EngineServer`. 12

[5] Bisque installation. `http://biodev.ece.ucsb.edu/projects/bisquik/wiki/InstallationInstructions05`. 13, 14

[6] Bisque mex request. `http://biodev.ece.ucsb.edu/projects/bisquik/wiki/Developer/ModuleSystem/MexRequestSpecification`. 4

[7] Bisque module creation tutorial. `http://biodev.ece.ucsb.edu/projects/bisquik/wiki/Developer/ModuleSystem/Tutorial`. 4, 7, 17

[8] Bisque module specification. `http://biodev.ece.ucsb.edu/projects/bisquik/wiki/Developer/ModuleSystem/BisqueModuleSpecification`. 4, 5

[9] Trellis: iplant user management. `https://user.iplantcollaborative.org`. 2