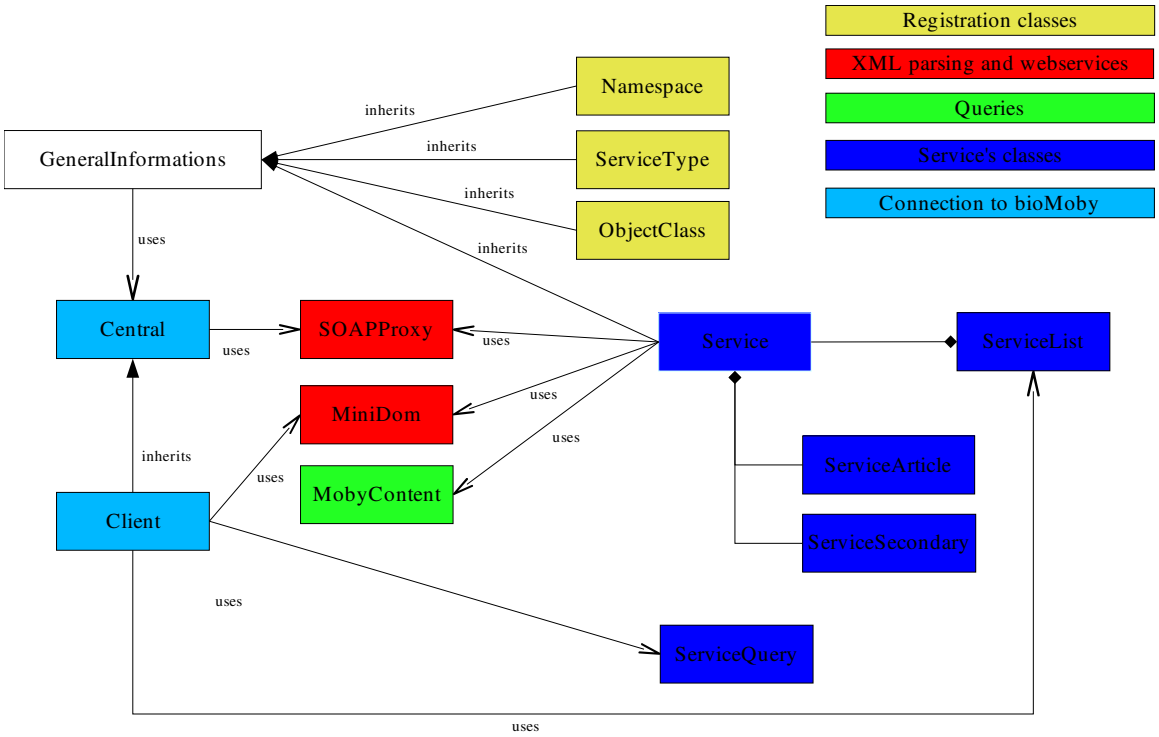
BioMoby Python

Version 0.6 10/22/2004

Classes



The kernel of the system is made of modules SOAPProxy and MiniDom.

The class Client describes a connexion to bioMoby and contains the assiociated methods (registration, queries, etc.)

The class Service wraps informations about the remote service and executes remote methods.

The class ServiceQuery builds queries for the method findService.

The class ServiceList manages lists of services, it is the result of method FindService.

Some classes doesn't appear in this diagram, these classes are related to Moby Object's ontology.

API Documentation

See the index.html in the <package_src_dir>/doc

How does it works?

First step, import modules:

In all your Moby Client applications you should add:

>>>from bioMoby import *

This will add all necessary classes in order to deal with bioMoby server and registrated services.

Set up a connection with the bioMoby server:

First declare a bioMoby client:

>>>client=Client()

by default, url of the server is http://mobycentral.cbr.ncr.ca/cgi-bin/MOBY05/mobycentral.pl.

You can specify another server with the url parameter:

>>>client=Client(url="http://anotherMobyServer")

You can also specify a different namespace for your mobyServer with the parameter ns:

>>>client=Client(url="http://anotherMobyServer/cgi-bin/server.pl", ns="http://anotherMobyServer//namespace)

Find services?

For example, you are looking for a service which name is getSequenceFromPDB:

>>>client.findService(serviceName='getSequenceFromPDB')

The command will return a ServiceList object that contains your service.

The following examples show how to find services with specific types:

```
>>>query=ServiceQuery(inputObjects=[
                        ('', 'DNASequence', ['NCBI_gi', 'NCBI_Acc']),
                        ])
>>>query=ServiceQuery(outputObjects'=[
                        ('', 'GenericSequence', [])
                        ])
>>>query=ServiceQuery(protocol='cgi',
                              outputObjects=[
                              ('GenericSequence','aObjectType', ['EMBL', 'PIR', 'NCBI_gi', 'NCBI_Acc'])
                              ])
```

As you can see, an article is described in a tuple: ('articleName', 'objectType', ['Namespace'])

after, we can retrieve our list of services:

>>>listOfServices=client.findservice(query)

The variable listOfServices will contain a list of services that matches your query. If the command is successful, the command len(listOfServices) should return at least one:

>>>len(listOfServices)

<number of services found>

We want to have some informations about the first service:

>>>service=listOfServices[0]

List of input objects:

>>>service.inputObjects

List of output objects:

>>>service.outputObjects

the XML of the service:

>>>str(service)

If you know the service's name and you want to use the service directly, you can retrieve its WSDL:

>>>service=Service(client.retrieveServiceWSDL('getSequenceFromPDB'))

The command will build a service from the given WSDL. Not all informations will be retrieved from the WSDL such as input/output objects and secondary parameters.

Prepare the parameters for the service:

As described in the Moby's API, all Moby's objects are XML strings:

<Object namespace=”a namespace” id=”an ID”>Content</Object>

Use the MobyMarshaller to translate your Python Object into a Moby XML Object:

>>>myObject=MyOwnMobyObject()

>>>m=MobyMarshaller()

>>>m.dumps(myObject)

If the command is successful, you should have the XML form of the object.

To use objects with services, we must build a content object:

>>>mobyContent=MobyContent({'query1':[myObject]})

Queries are always lists.

To add optional parameters for the query, specify them with the Parameter object:

>>>param1=Parameter('threshold': 1.24)

>>>mobyContent=MobyContent({'query1':[aPDB, param1]})

A few words about the MobyMarshal module:

The MobyMarshaller object will translate a Python Object into its XML form, the MobyUnmarshaller object do the contrary.

You can build your own Moby(Un)Marshaller Object (and it is recommanded), all you have to do is to build a subclass and override the dumps method (for the MobyMarshaller class) and the loads method(for the MobyUnmarshaller.

This is how the MobyUnmarshaller class works:

>>>um=Unmarshaller()

>>>o=um.loads(xml)

If you want to make your own class, you'll need to specify two methods:

- toMoby:it will serialize your object in XML
- fromMoby it will deserialize the XML

Execution of services

MobyContent is now a Moby Content Object, send it to your remote bioMoby service and get the result:

```
>>>result=service.execute(mobyContent)
```

The variable result is a XML string but it can be reused with another service as the method “execute“ makes no differences between the XML string and a MobyContent object:

```
>>>result1=service1.execute(result)
```

Or if you want further manipulations, you can first translate it into a MobyContent object by using a MobyUnmarshaller Object:

```
>>>um=MobyUnmarshaller()
```

```
>>>mc=um.loads(result)
```

the variable mc will contain a MobyContent object.

From the 0.5 version, execute can return a MobyContent Object directly without having to create a MobyUnmarshaller object:

```
>>>result=service.execute(mobycontent, returnXml=False)
```

This will return a MobyContent object.

From the 0.6 Version, you can trace SOAP calls between your script and the service by using the debug option:

```
>>>result=service.execute(mobycontent, debug=True)
```

The execute method will print an output like this:

```
In build.
In dump. obj= <?xml version="1.0" encoding="UTF-8"?><moby:MOBY
xmlns:moby="http://www.biomoby.org/moby"><moby:mobyContent><moby:mobyData
moby:queryID='query1'><moby:Simple><moby:Object moby:namespace="AGI_LocusCode" moby:id="At3g19100"
moby:articleName=""/></moby:Simple></moby:mobyData></moby:mobyContent></moby:MOBY>
In dump_string.
In gentag.
In dumper.
*** Outgoing HTTP headers **********************************************
POST /cgi-bin/moby_services/Services/genomic_services/genomic_services.cgi HTTP/1.0
Host: arabidopsis.info
User-agent: SOAPpy 0.11.4 (http://pywebsvcs.sf.net)
Content-type: text/xml; charset="UTF-8"
Content-length: 904
SOAPAction: "http://biomoby.org/#getNASC_codebyAGI_locus"
************************************************************************
*** Outgoing SOAP ******************************************************
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" xmlns:SOAP-
ENC="http://schemas.xmlsoap.org/soap/encoding/" xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance" xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/" xmlns:xsd="http://www.w3.org/1999/XMLSchema">
<SOAP-ENV:Body>
<ns1:getNASC_codebyAGI_locus xmlns:ns1="http://biomoby.org/" SOAP-ENC:root="1">
<v1 xsi:type="xsd:string">&lt;?xml version="1.0" encoding="UTF-8"?&gt;&lt;moby:MOBY
xmlns:moby="http://www.biomoby.org/moby"&gt;&lt;moby:mobyContent&gt;&lt;moby:mobyData
moby:queryID='query1'&gt;&lt;moby:Simple&gt;&lt;moby:Object moby:namespace="AGI_LocusCode" moby:id="At3g19100"
moby:articleName=""/&gt;&lt;/moby:Simple&gt;&lt;/moby:mobyData&gt;&lt;/moby:mobyContent&gt;&lt;/moby:MOBY&gt;</v1>
</ns1:getNASC_codebyAGI_locus>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
************************************************************************
code= 500
msg= Internal Server Error
headers= Date: Fri, 22 Oct 2004 07:36:41 GMT
Server: Apache/2.0.40 (Red Hat Linux)
Content-Length: 617
```

Connection: close
Content-Type: text/html; charset=iso-8859-1

content-type= text/html; charset=iso-8859-1
data= <!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
<title>500 Internal Server Error</title>
</head><body>
<h1>Internal Server Error</h1>
The server encountered an internal error or
misconfiguration and was unable to complete
your request.
Please contact the server administrator,
 david@arabidopsis.info and inform them of the time the error occurred,
and anything you might have done that may have
caused the error.
More information about this error may be available
in the server error log.
<hr />
<address>Apache/2.0.40 Server at arabidopsis.info Port 80</address>
</body></html>

*** Incoming HTTP headers **
HTTP/1.? 500 Internal Server Error
Date: Fri, 22 Oct 2004 07:36:41 GMT
Server: Apache/2.0.40 (Red Hat Linux)
Content-Length: 617
Connection: close
Content-Type: text/html; charset=iso-8859-1
****

Small scripts in the utils directory.

The moby2python script

You'll find in the <bioMoby-python directory>/utils a small script that generates a Python class from a bioMoby XML definition.

With no argument:

$moby2python

In the output directory, you'll find all the registrated classes converted in different files, for example SchematikonStructureAnnotation.py contains a MobySchematikonStructureAnnotation object.

With arguments:

$moby2python <a directory> Object1 Object2 Object3

A module will be generated with only registrated Object1 Object2 and Object3.

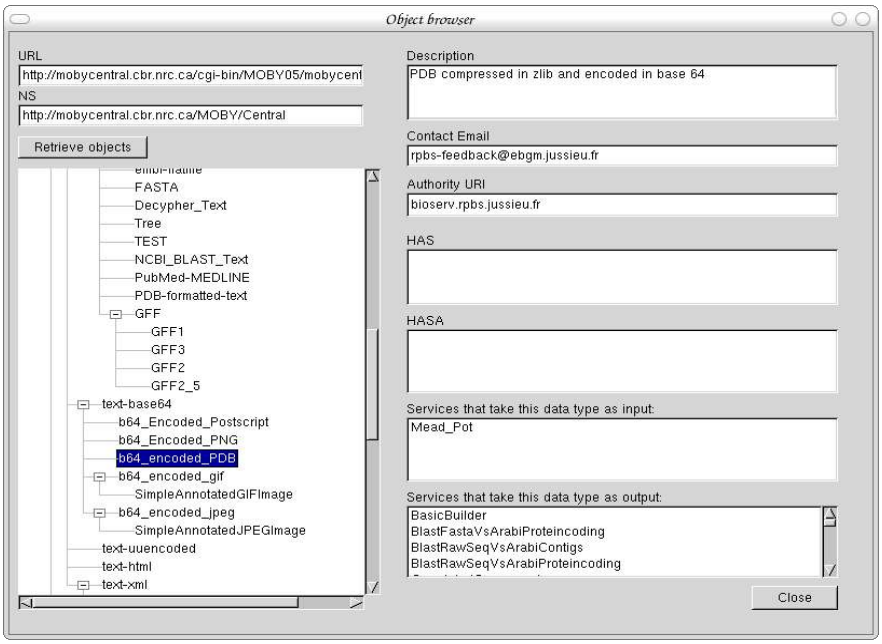
If you want to use these classes with the MobyMarshal module, you must put them in the <bioMoby-python>/ontology directory. Thus, classes are managed like any other Python's object.

This can be useful when you want to link Moby objects with BioPython's objects.

The objectBrowser script:

In order to run the script, you'll need to have the wxPython package installed. It can be retrieved on http://sourceforge.net/project/showfiles.php?group_id=10718.

This script represents the bioMoby ontology in a Tree:



In order to retrieve the ontology, click on the “Retrieve objects” button. You'll see (after some time) the tree representing the ontology of bioMoby.

Select a node in the tree view in order to have more informations about the object.

Register services and objects with bioMoby-python

Register objects:

First set up informations about your object:

```
>>>zmPDB=ObjectClass(
   authURI='prot3.rpbs.jussieu.fr',
   contactEmail='yanwong@ebgm.jussieu.fr',
   description='a WIP of a compacted and encoded PDB file',
   objectType='zPDB',
   relationships=[('ISA',{'':'text-base64'})]
   })
```

then register:

```
>>>result=zmPDB.register()
```

To see if you are lucky:

```
>>>result.isSuccess()
```

If something went wrong during the process, you can have the message delivered by bioMoby:

```
>>>result.message
```

An object can be registrated even if it does not exist as an implemented class!

Relationships are described in a list of dictionary, for example the list tells that the entity is an object and has strings:

```
[('ISA', {'': 'Object'}),('HAS',{'SEQ':'String', 'STR':'String'})]
```

Register services:

First build your service:

```
service=Service(
   {'category':'moby',
    'name':'getStrideFromPDB',
    'type':'Analysis',
    'url':'http://prot3.rpbs.jussieu.fr:8080/ZStrideService/ZStrideService.py',
   'signatureURL':'http://prot3.rpbs.jussieu.fr:8080/RDF',
    'authoritative':0,
    'contactEmail':'yanwong@ebgm.jussieu.fr',
    'authURI':'prot3.rpbs.jussieu.fr',
    'description':'This is a service that uses stride, it takes as argument a compressed PDB file',
    'inputObjects':[ServiceArticle(name='', type='zPDB',namespaces=[])],
    'outputObjects': [ServiceArticle(name='', type='StrideOutput',namespaces=[])]
    })
```

inputObjects, outputObjects must be described using ServiceArticle object, if you need to specify a collection of ServiceArticle, put a list of ServiceArticle objects:

```
{'collectionName':[ServiceArticle, ServiceArticle]}
```

After the registration is completed, you'll have to save the resulting RDF in the location where the bioMoby agent can find it:

>>result=service.register()

>>result.RDF

<The RDF of the registrated service>

How to build bioMoby web services

The bioMoby web service API:

The main class of this part is the Dispatcher Class. It is built upon the Multithread Class and allow simultaneous treatment of a MobyContent object as the MobyContent object is made of several independent queries.

The Dispatcher class needs an Invocator class (it makes call to applications, database or whatever treatment you want) a formatter function (it does the formatting results (transform raw results into Moby objects for example) and a toParameters function (that translate Moby queries into Invocator's parameters)

In order to ease or convert already existing applications into webservices, the package provides some prebuild Classes:

- LocalInvocator: use a command line application as a data provider

  It takes a tuple as argument:

  (commandName as String, commandParameter as String, [temporary files], [outputfiles])

  return the result of the call of the command line (either it takes all output files or the stdout)

- CGIPostInvocator: use a CGI application as a data provider

  return the result of the call of the CGI script

Of course you can build your own Invocator class, the only thing you have to provide is a execute() method.

These classes intended to help you to write webservices from CLI applications or simple CGI scripts. However, I strongly recommand to do things differently!

Build a web service from a CLI application

Here an example built upon Stride. Stride take as argument a PDB and return a formatted text as a result:

```
        def _toParameters(queryData):
            if len(queryData)!=1:
                raise Exception, "A single PDB object/file is needed"

            datum=queryData[0]

            from bioMoby import MobyObject, MobyZmPDB
            import random, urllib

            commandName="/application/bin/stride"

            parameters=[]
            tempfiles=[]


            tfs='/tmp/tmp'+ `int(random.random()*1000000)`

            if datum.__class__ is MobyObject and datum.namespace=="PDB":
                u=urllib.urlopen
        ("http://www.rcsb.org/pdb/cgi/export.cgi/"+datum.id+"?format=PDB&pdbId="+datum.id+"
        &compression=None")
                f=file(tfs,"w")
                f.write(u.read())
                u.close()
                f.close()
            elif 'content' in dir(datum) and '_articleName' in dir(datum):
                fp=file(tfs,"w")
                fp.write(datum.content)
                fp.close
```

```
        parameters.append("-f "+tfs)
        tempfiles.append(tfs)

        return (commandName, "".join(parameters), tempfiles)

    def stride(mobyContent):
        """Dispatch the contents into a pool of invocators
        """
        from bioMoby import Dispatcher, LocalInvocator

        d=Dispatcher(mobyContent, LocalInvocator, __toParameters)

        return d.execute()

    from ZSI import dispatch
    dispatch.asCGI()
```

The dispatcher does all the job, translating (thanks to the __toParameters function) the MobyObjects into CLI arguments and sending parameters to the LocalInvocator class.

This example show you how to implement a web service from a CLI that return outputfiles:

```
    #Clustalw
    def _toParameters(queryData):
        commandName="clustalw"
        parameters=[]
        tempfiles=[]
        outputfiles=[]

        from bioMoby import Parameter
        import random, string

        tfs='/tmp/tmp'+ `int(random.random()*1000000)`

        for datum in queryData:
            if datum.__class__ is Parameter:
                parameters.append("-"+datum.articleName+"="+datum.value)
            else:
                try:
                    fp=file(tfs,"w")
                    fp.write(datum.content)
                    fp.close
                    parameters.append("-infile="+tfs)
                    tempfiles.append(tfs)
                except:
                    pass

        tempfiles.append(tfs+".dnd")
        outputfiles.append(tfs+".aln")

        return (commandName, " type=p -align "+string.join(parameters," "), tempfiles,
    outputfiles)

    def clustalw(mobyContent):
        from bioMoby import Dispatcher, LocalInvocator

        d=Dispatcher(mobyContent, LocalInvocator, _toParameters)

        return d.execute()

    from ZSI import dispatch
    dispatch.asCGI()
```

Asynchronous webservices

Unlike synchronous webservices, asynchronous web services don't return immediately the results! Sometimes treatment can be lonb, the webserver can't hold a connection quite long and will probably cut the connection, the Client interface will then return a timeout exception.

In order to solve this issue, the dispatcher return any results within the two minutes, either it returns

the results of the requests or it will return an MobyObject with a JOBSESSION namespace.

If you receive an object with a JOBSESSION, this probably means that your treatment will last more that two minutes!

It is then up to your script to make an active wait. Resend the MobyContent object with your JOBSESSION in order to retrieve results. The dispatcher will look for the results according to the request, if it can't retrieve results, it will send the request back.

Note that the client interface of the bioMoby-python API doesn't manage automatic recall to asynchronous webservices (the program has to do it by himself).

How does work the Dispatcher:

It first receives an XML representing the MobyContent object. It then splits the MobyContent in several queries.

Queries are then transformed into Invocator's parameters via the _toParameters function.

The queries are then treated with a pool of thread and the Invocator Class.

Results of Queries are then sent to the formatter function. This function tranforms raw results sent by Invocator Objects into a list of Moby Objects.

The final MobyContent Object is sent back when ALL queries are treated.

Examples:

Sources can be found in <packages_src_dir>/tutorials

tutorials/MobyCentral: how to deal with the Moby Central server:

registerService.py: Show how to register a service

retrieveServices.py: Show how to retrieve services name and description

servicesDump.py: A small program that dump all informations about services

tutorials/Service: how to deal with Moby Services:

async.py: How to make asynchronous call to a webservice.

testMarshaller.py: How to use the mobyMarshal module

testMobyService.py: from a Perl Example, however, it uses mobyMarshal module to read data

testMobyService2.py: from a Perl Example (the one with flowers)

tutorials/webservice: An example of how to build webservices with the bioMoby Python API