OmniGene Platform Overview

Introduction

OmniGene is a platform and collection of frameworks that provide an infrastructure and "glue" that enable disparate development teams, working in different languages to leverage each other's work. OmniGene is specifically built for the bioinformatician who must deal with large, distributed, and often ambiguous data and data types. OmniGene is implemented using Java Enterprise technology and web services technologies including Enterprise Java Beans (EJB), XML, and Simple Object Access Protocol (SOAP). The OmniGene team understands that each institution and development team has their own expertise and engineering requirements in terms of: level of development experience, engineering needs, time constraints and IT infrastructure. Therefore, the we have tried to abstract away as much of the "hard stuff" as possible leaving the Bioinformatician to concentrate on the task at hand. In the following few sections we will try and guide the beginning OmniGene developer through the installation, set-up, and runtime environment required to run your very own OmniGene instance. we will start with a little historical background, and then quickly transition to the nuts and bolts of the platform.

A Little History

The OmniGene platform is a 3+ year old project that has gone through many rewrites and transitions. OmniGene is sponsored and originally funded by the Whitehead Institute, an organization dedicated to scientific endeavors such as:

The sequencing of the Human Genome
Comparative Genomics
Medical and Population Genetics
Cancer Research

For more information please visit our website at: http://www-genome.wi.mit.edu this site will give a good overview of our work and vision for genomic, proteomic, medical, and cancer research.

These scientific endeavors consume a vast quality of programmer resource and IT infrastructure. In fact, the Whitehead Institute current employs over 70 Bioinformaticians and more than 350 laboratory technicians. They have produced *massive* quantities of data that must be put into the public domain and analyzed to find out what makes us tick, where we came from, why we are predisposed to certain diseases and why certain cells go awry and become cancerous. The questions do not stop there as there are many, many more questions that would fill this document.

A few years ago Bioinformatics was a nascent field, there were no O'Reilly books that described how to perform Blast, search databases, or how to wield the Perl programming language to solve "common" Bioinformatics tasks. Bioinformaticians had (and still has) very variable backgrounds and skills from: the Biologist who had learned programming out of a book, to the computer scientist well versed in algorithmic design and analysis. Somewhere in the middle of these two programmer types lies the software engineer who must,: aggregate data from disparate databases, run algorithms for users, and provide user interfaces that suite the scientific needs of their Biologist counterpart (read customer). These people are the support system or "glue" that is quickly becoming the foundation of Biology today. OmniGene was first built specifically for these programmers as we observed that many of the programming tasks that these people were performing were redundant across groups and approximately 40-50% of their time could be saved if these tasks could be abstracted into reusable software components. Below is a list of some of

the activities that OmniGene aims to produce as reusable components or contracts for programmers to follow:

1) Data visualization in terms of *genomic* information
2) Data aggregation from disparate public/private databases
3) Asynchronous execution of command line driven tools
4) Semantic correlations between data and data types

We also quickly realized that, the level of skill required to produce and consume data was very variable in the Bioinformatics space. Herein lies the ultimate challenge: To produce a software framework that was accessible to the Bioinformatics community at large without making the community have to jump through hoops to understand and start seeing results.

We believe that we have produced such a framework and runtime environment using tried and true software engineering principles garnered from other software engineering communities such as: Financial and Business domain, Enterprise Java infrastructure groups (Jboss etc) and Physics/Mathematics engineering domain.

As you can see, OmniGene is quite an ambitious project with many different, yet complimentary, components. However, we feel that, by cross cutting the Bioinformatics domain into these 4 activities, a complete software architecture will be produced that is capable of satisfying our target audience: The Genomics or Discovery based researcher.

High Level Concepts

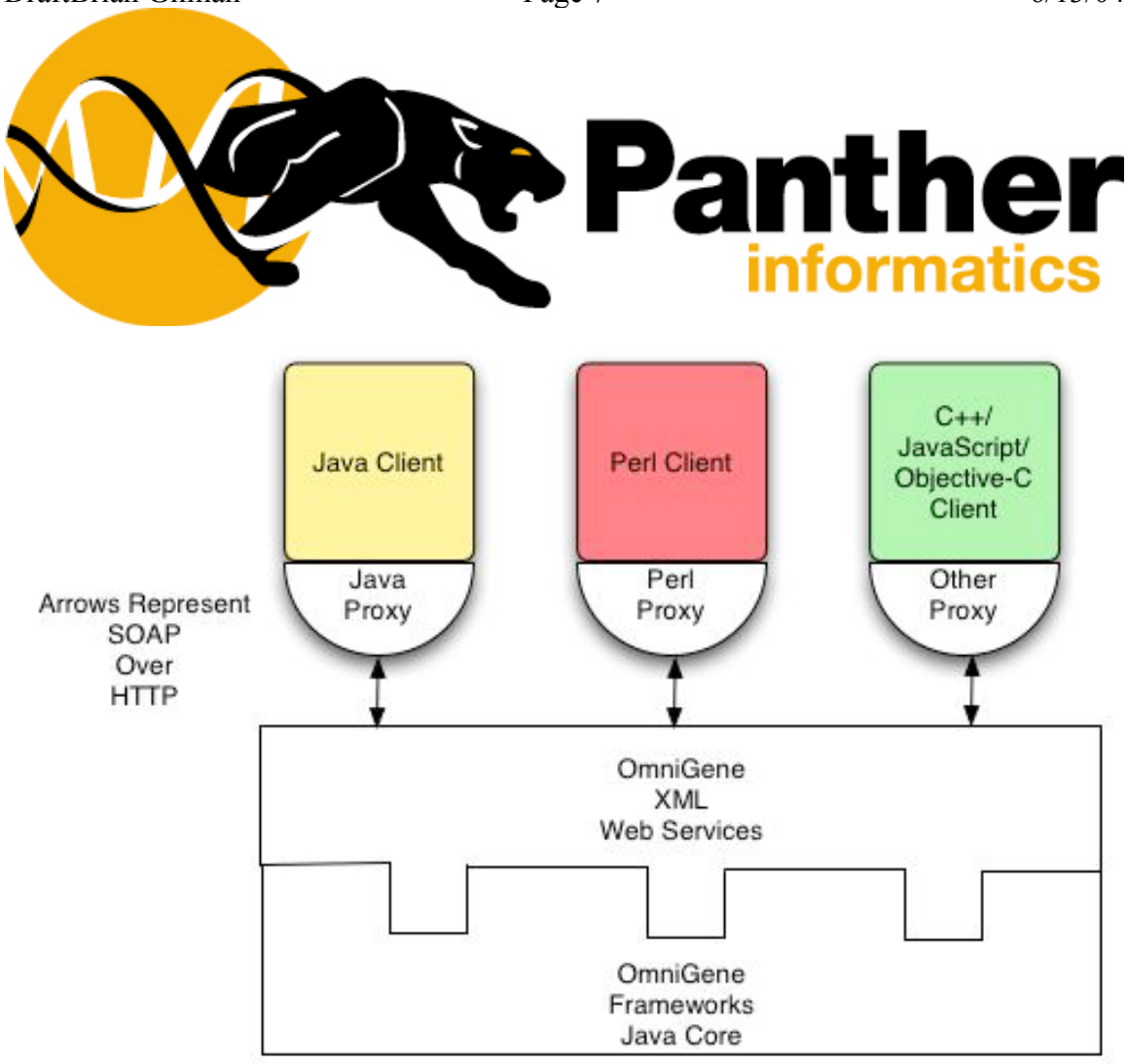
Design Goals:

OmniGene's main design goal is to allow as many programmers access to the services that the middleware exposes. OmniGene calls the components that it exposes **Services**. **Services** are built upon software templates called **Frameworks**. We will explore more of the design of **Services** and **Frameworks** in the section: **OmniGene Framework Overview** and **OmniGene Directory and Code Structure**.

Design Methodology:

OmniGene utilizes the Java programming language at its core and exposes its capabilities as web services. This allows client applications, written in a multitude of different languages, to access these services without having to "know" the internals of the OmniGene engine. This is reflected in our CVS tree in fact, developers have written client applications in Python, Perl, Objective-c and Javascript (with others coming). In fact, any languages that supports web services (XML over http using the SOAP dialect) may access our java core. Developers looking to extend the engine must gain some familiarity with the core components and subsystems that are documented within this manual.

Figure 1 shows a graphical depiction of how the platform exposes **Services** to a heterogeneous programming environment:



Clients wishing to access services can either utilize **Proxy** code that is provided as part of the framework or write their own **Proxy** to OmniGene web services. The section:" The Web Service Framework" provides an overview of how to utilize an OmniGene **Proxy** or write your own.

OmniGene Framework Overview:

Figure 2 shows a graphical depiction of the omnigene frameworks. They are shown in the vertical box labeled **Framework Layer.** As mentioned before, **Frameworks** are templates that the Bioinformatician must adhere to writing their own **Service**. Frameworks provide functionality such as:

1) Exposure of a piece of code as a web service
2) Connections to a biological data store
3) Handling of messages from clients and dispatch to the appropriate piece of middleware
4) Providing security and authentication schemes
5) Finding other pieces of middleware on the network

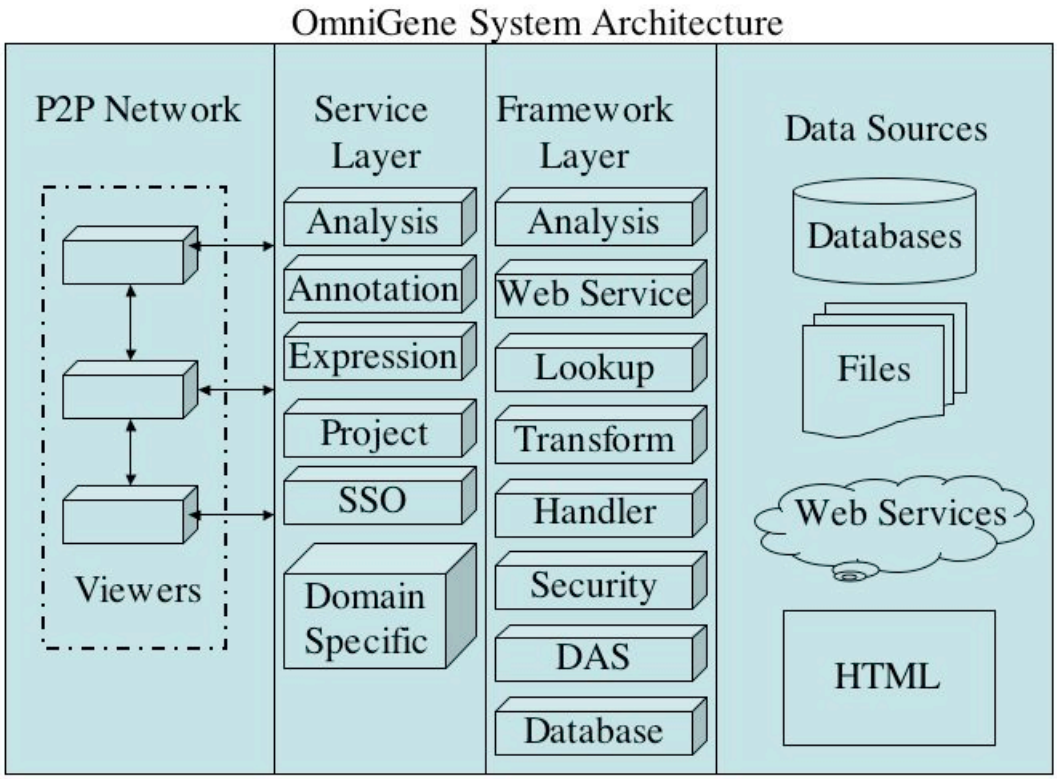


Figure 2: The OmniGene System Architecture

Getting To Know OmniGene Frameworks:

As stated above OmniGene Frameworks are software templates or contracts that the software developer must adhere. These design contracts allow the middleware to manage the life cycle of your component. This includes: instantiation, caching, clean up, and destruction. By following the design contracts outlined in the following sections you will never have to think about the life cycle of your component. Instead, you can concentrate on the problem at hand and clever solutions to that problem.

In this section we will dive deep into each framework component, the contracts you must follow, how each framework works, its design strategy, dependencies on other frameworks or third party tools and its life cycle. We will conclude each section with a piece of example code that utilizes this piece of the framework and bring you through the example line by line.

The Analysis Framework:

Introduction:

Releasing algorithms to the biological community (or any research environment) is typically a painstaking and error prone process. This stems from the fact that bioinformaticians do not usually write their programs with a user interface in mind and, instead, depend on a command line interface.

Typically, command line interfaces satisfy only a subset of so-called power users in an organization, leaving others in the dark in terms of input parameters, file formats, and output generated from these programs. READMEs and other documentation are usually left unread, forcing the developer of the program to answer the same questions time and time again.

Worse still, once a command line program becomes popular to the not so tech savvy biologist, the bioinformatician is left running their program each and every time data is ripe for analysis. One common solution to this problem is to set up a web interface for the popular program and let the biologist run it for themselves. This works well for a single program but becomes quite cumbersome when there are hundreds.

The OmniGene Analysis Engine (OAE) solves this problem by providing a runtime engine that interfaces with any command line driven program by exposing these programs as web services using a single common interface and a little glue code. The

algorithm developer writes a single client application in their favorite programming language and makes this available to their biologist counterpart. After the interface has been produced the algorithm developer is freed from having to run their programs for the biologist. The bioinformatician can then concentrate on writing programs that solve the next set of interesting biological problems.

Analysis Framework Design:

The OmniGene analysis framework was designed to allow any command line tool written in any language to be exposed as a web service. Figure 3 shows the general design of the analysis framework in terms of its runtime components.

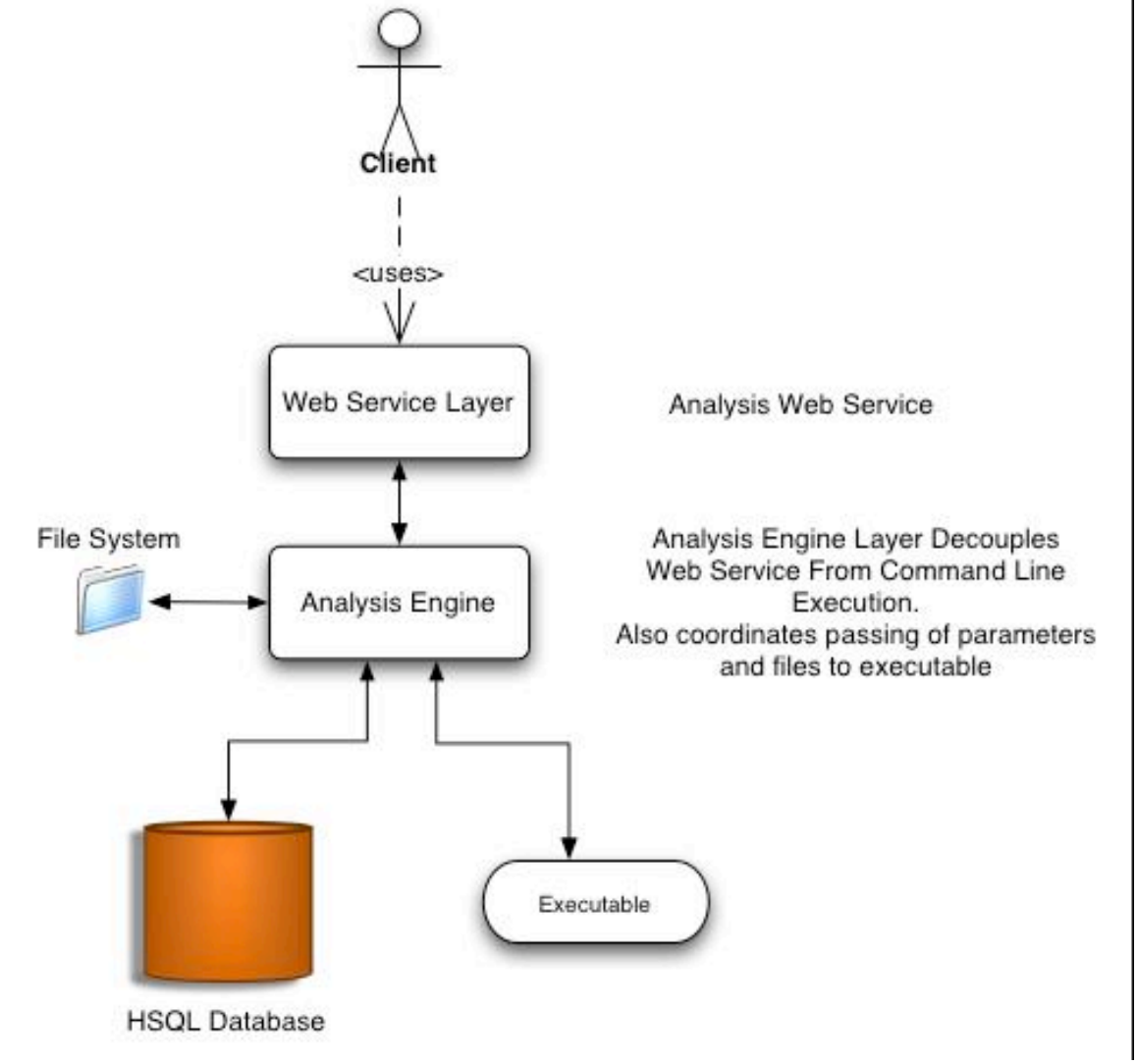


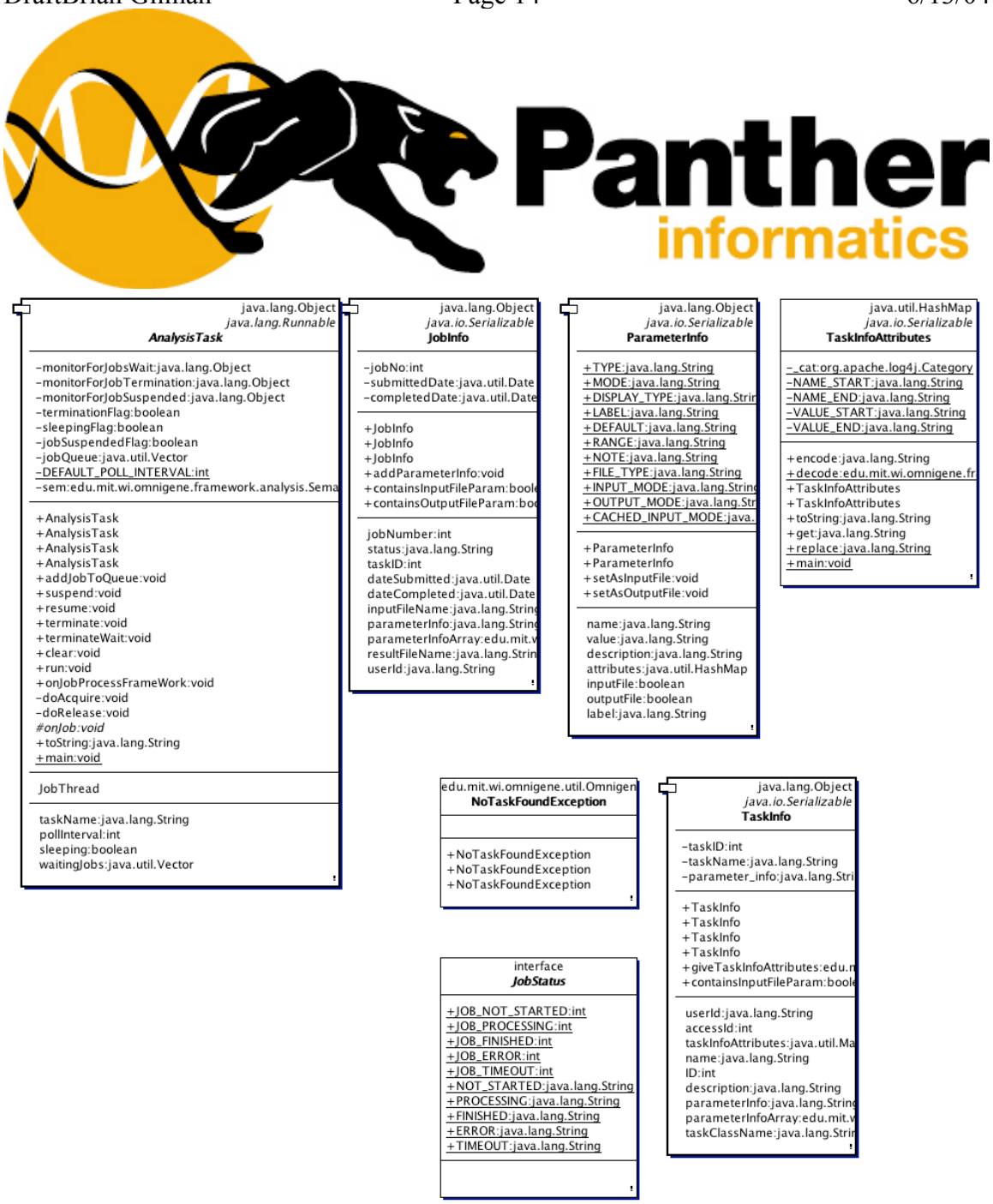
Figure 3: Analysis Framework Runtime Overview

Table 1: Enumeration of the components found in Figure 3 and provides a brief description of how they are utilized in the platform:

Component Name	Description
Analysis Web Service	Provides an interface to the Analysis Engine. Provides SOAP interface to clients
Analysis Engine	Keeps track of files in the database, executable status (running/dead/etc), output files, and executable life cycle
HSQLDB	Used as data store for status information, file location, as well as other Task information
File System	Used to store output files from your command line tool
Executable	Any executable written in any language

The OmniGene Analysis Engine is written as a set of EJB's and is deployed in the JBoss Enterprise Java Bean Container. The Analysis Web Service utilizes Apache's Tomcat Servlet Container and the Apache Axis SOAP engine. HSQLDB is used so that this engine may exist on any hardware platform. Previous versions of this engine and framework utilized the Postgresql database however, this choice of RDBMs excluded Windows users as cygwin was needed to perform its installation.

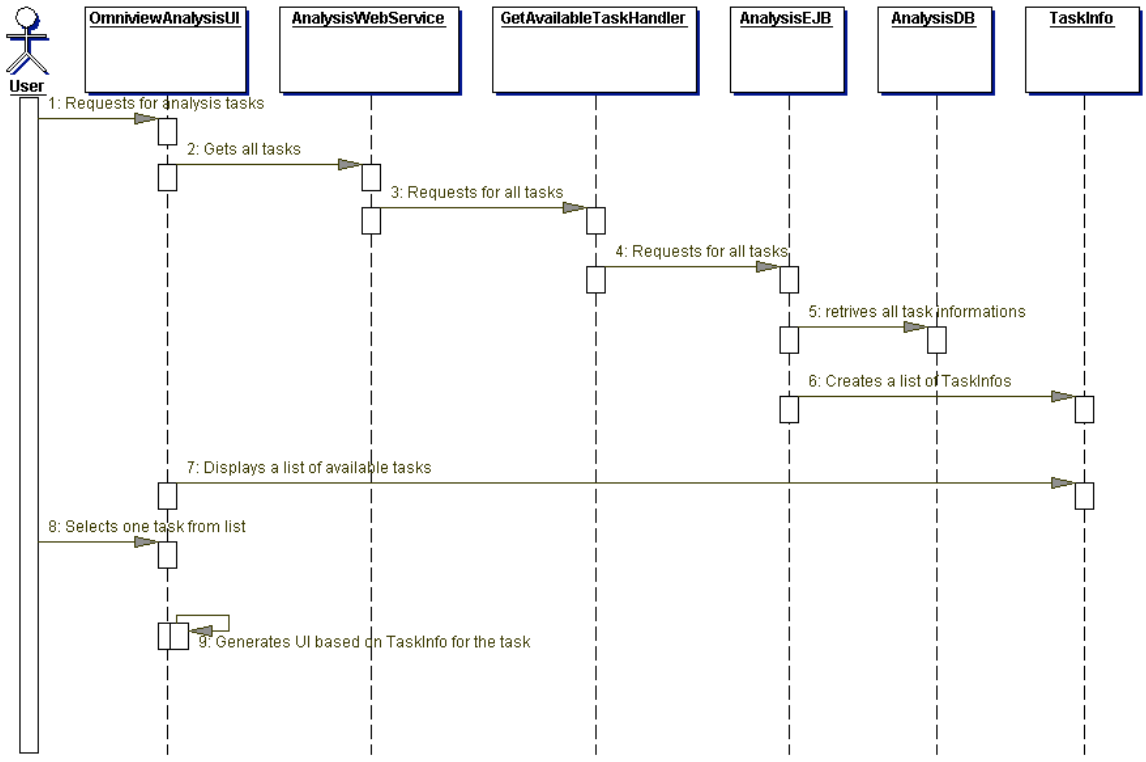
Main Framework Classes:


java.lang.Object
java.lang.Runnable
AnalysisTask
-monitorForJobsWait:java.lang.Object
-monitorForJobTermination:java.lang.Object
-monitorForJobSuspended:java.lang.Object
-terminationFlag:boolean
-sleepingFlag:boolean
-jobSuspendedFlag:boolean
-jobQueue:java.util.Vector
-DEFAULT_POLL_INTERVAL:int
-sem:edu.mit.wi.omnigene.framework.analysis.Sema
+AnalysisTask
+AnalysisTask
+AnalysisTask
+AnalysisTask
+addJobToQueue:void
+suspend:void
+resume:void
+terminate:void
+terminateWait:void
+clear:void
+run:void
+onJobProcessFrameWork:void
-doAcquire:void
-doRelease:void
#onJob:void
+toString:java.lang.String
+main:void
JobThread
taskName:java.lang.String
pollInterval:int
sleeping:boolean
waitingJobs:java.util.Vector
java.lang.Object
java.io.Serializable
JobInfo
-jobNo:int
-submittedDate:java.util.Date
-completedDate:java.util.Date
+JobInfo
+JobInfo
+JobInfo
+addParameterInfo:void
+containsInputFileParam:boole
+containsOutputFileParam:boo
jobNumber:int
status:java.lang.String
taskID:int
dateSubmitted:java.util.Date
dateCompleted:java.util.Date
inputFileName:java.lang.String
parameterInfo:java.lang.String
parameterInfoArray:edu.mit.v
resultFileName:java.lang.Strin
userId:java.lang.String
java.lang.Object
java.io.Serializable
ParameterInfo
+TYPE:java.lang.String
+MODE:java.lang.String
+DISPLAY_TYPE:java.lang.Strir
+LABEL:java.lang.String
+DEFAULT:java.lang.String
+RANGE:java.lang.String
+NOTE:java.lang.String
+FILE_TYPE:java.lang.String
+INPUT_MODE:java.lang.String
+OUTPUT_MODE:java.lang.Str
+CACHED_INPUT_MODE:java.
+ParameterInfo
+ParameterInfo
+setAsInputFile:void
+setAsOutputFile:void
name:java.lang.String
value:java.lang.String
description:java.lang.String
attributes:java.util.HashMap
inputFile:boolean
outputFile:boolean
label:java.lang.String
java.util.HashMap
java.io.Serializable
TaskInfoAttributes
-_cat:org.apache.log4j.Category
-NAME_START:java.lang.String
-NAME_END:java.lang.String
-VALUE_START:java.lang.String
-VALUE_END:java.lang.String
+encode:java.lang.String
+decode:edu.mit.wi.omnigene.fr
+TaskInfoAttributes
+TaskInfoAttributes
+toString:java.lang.String
+get:java.lang.String
+replace:java.lang.String
+main:void
edu.mit.wi.omnigene.util.Omnigen
NoTaskFoundException
+NoTaskFoundException
+NoTaskFoundException
+NoTaskFoundException
interface
JobStatus
+JOB_NOT_STARTED:int
+JOB_PROCESSING:int
+JOB_FINISHED:int
+JOB_ERROR:int
+JOB_TIMEOUT:int
+NOT_STARTED:java.lang.String
+PROCESSING:java.lang.String
+FINISHED:java.lang.String
+ERROR:java.lang.String
+TIMEOUT:java.lang.String
java.lang.Object
java.io.Serializable
TaskInfo
-taskID:int
-taskName:java.lang.String
-parameter_info:java.lang.Stri
+TaskInfo
+TaskInfo
+TaskInfo
+TaskInfo
+giveTaskInfoAttributes:edu.n
+containsInputFileParam:boole
userId:java.lang.String
accessId:int
taskInfoAttributes:java.util.Ma
name:java.lang.String
ID:int
description:java.lang.String
parameterInfo:java.lang.Strin
parameterInfoArray:edu.mit.v
taskClassName:java.lang.Strir

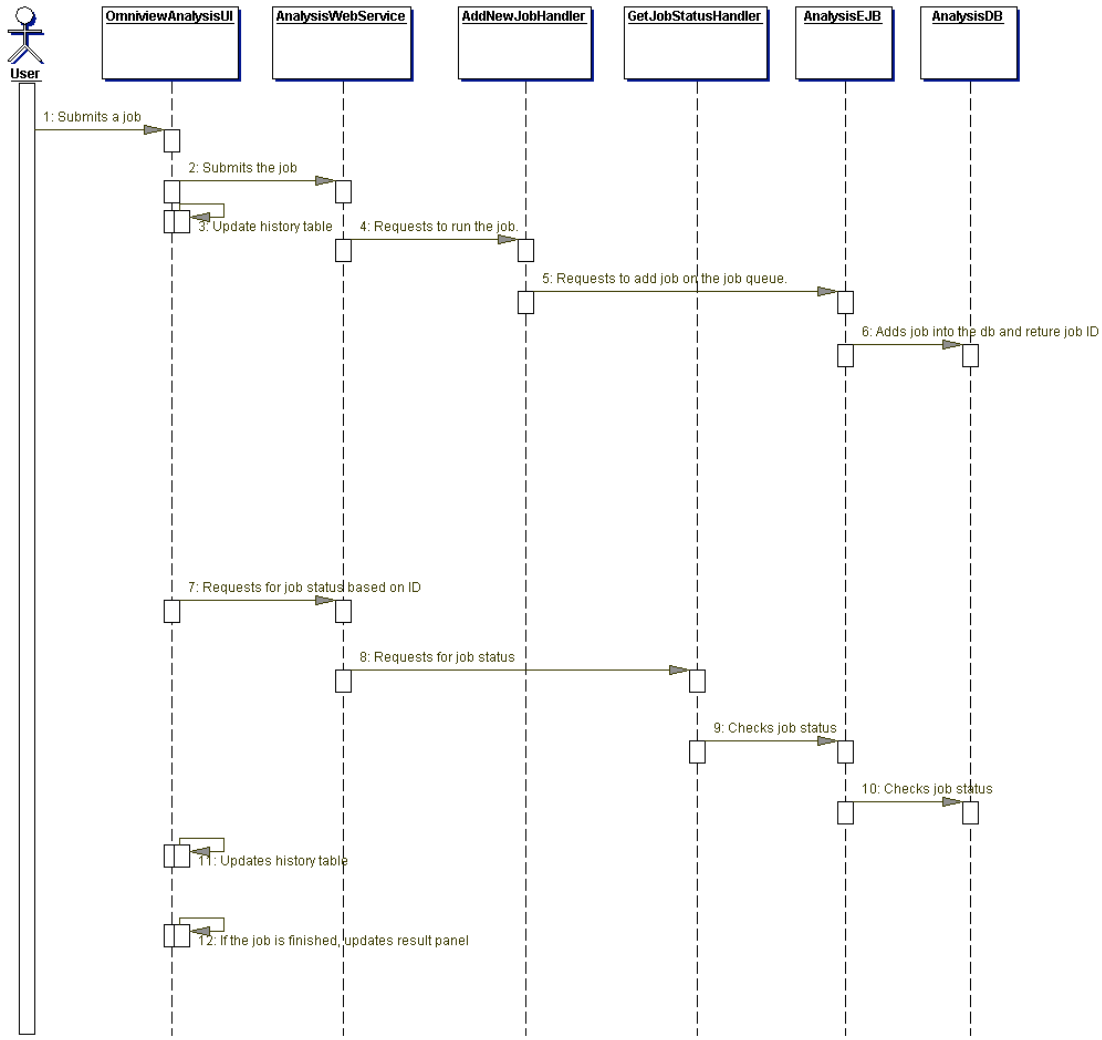
Class Name	Class Description
Analysis Task	This is the wrapper class which you must extend in order to execute your command line tool
JobInfo	Class that provides information about a particular instance of a running analysis task
ParameterInfo	Class that describes a particular parameter to your tool: name, whether it is a file or not etc
TaskInfo	Class that fully described your tool by encapsulating Parameters to your tool, name of the task etc.
TaskInfoAttributes	Bean that contains a collection of user defined attributes for this service. It allows you to push all attributes needed to run a task into the database
JobStatus	Enumeration Interface that describes the state of your running Job
Exceptions	When Something Goes Wrong and you've asked for a task we don't know, or asked about a Job you didn't submit to the engine etc. about we throw this

Analysis Framework Sequence Diagrams:

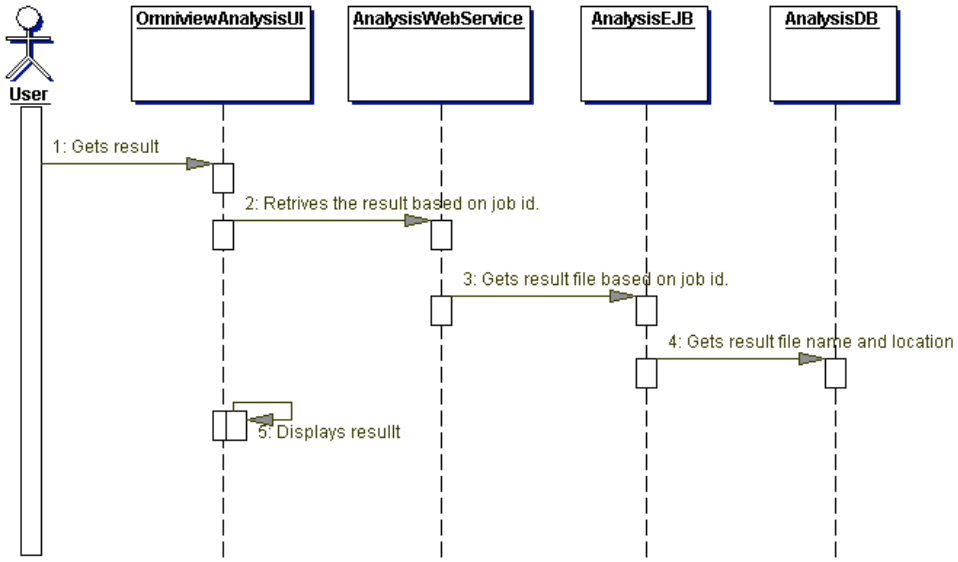
1. Get Tasks



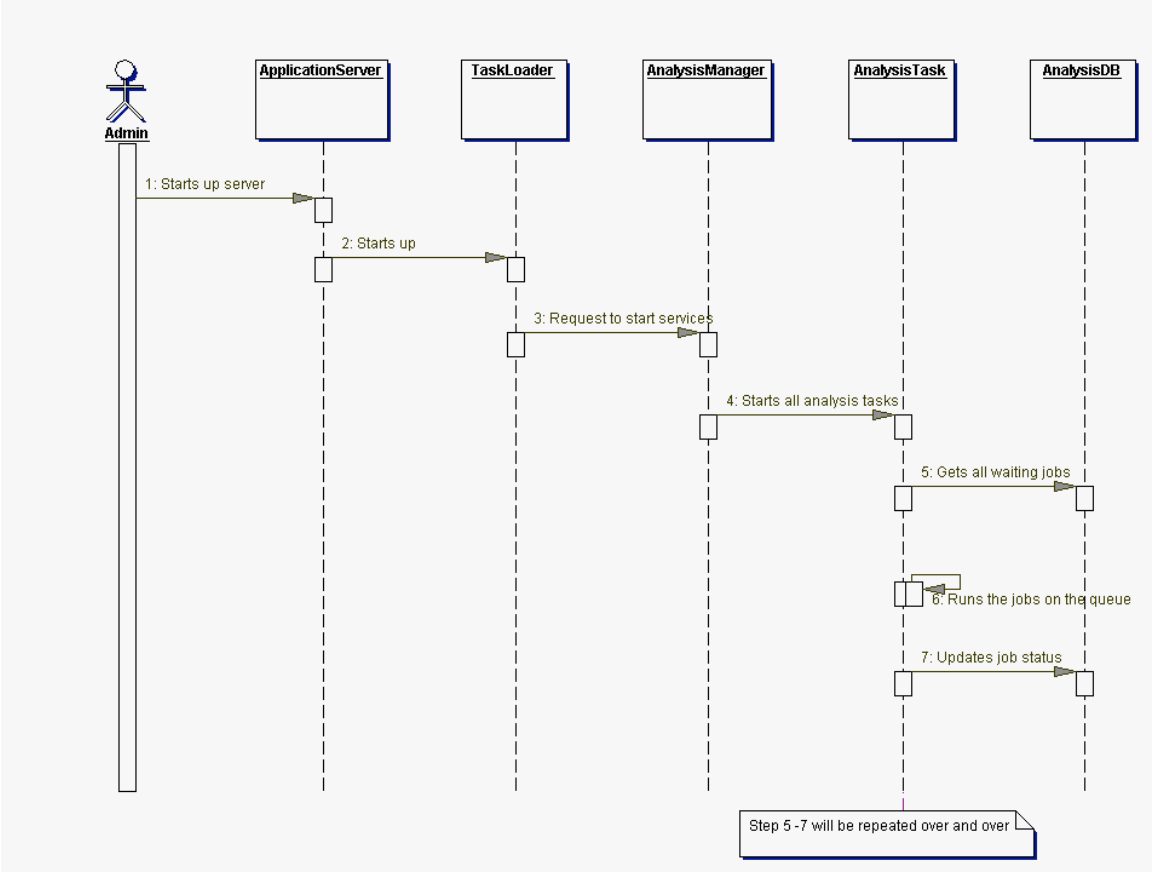
2. Submit job


User
OmniviewAnalysisUI
AnalysisWebService
AddNewJobHandler
GetJobStatusHandler
AnalysisEJB
AnalysisDB
1: Submits a job
2: Submits the job
3: Update history table
4: Requests to run the job.
5: Requests to add job on the job queue.
6: Adds job into the db and reture job ID
7: Requests for job status based on ID
8: Requests for job status
9: Checks job status
10: Checks job status
11: Updates history table
12: If the job is finished, updates result panel

3. Get result

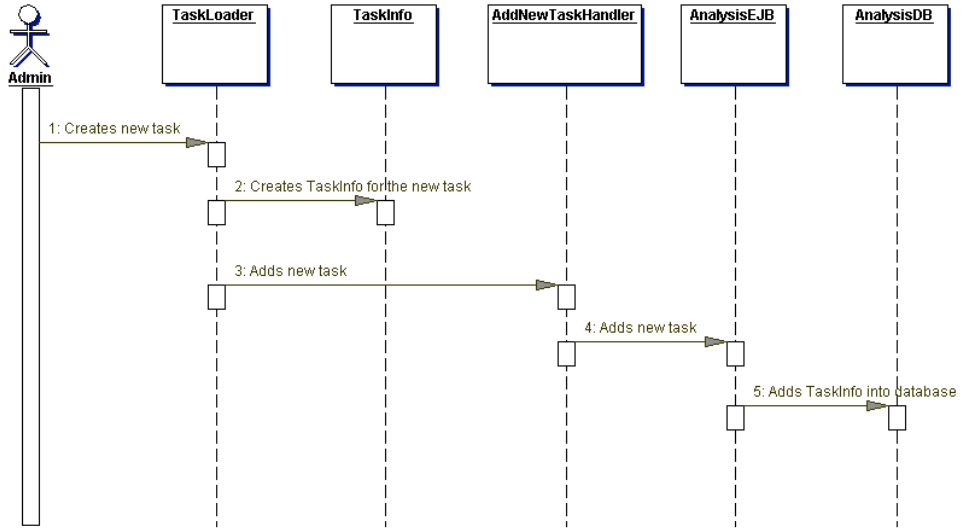


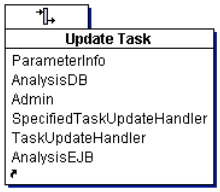
4. Running job



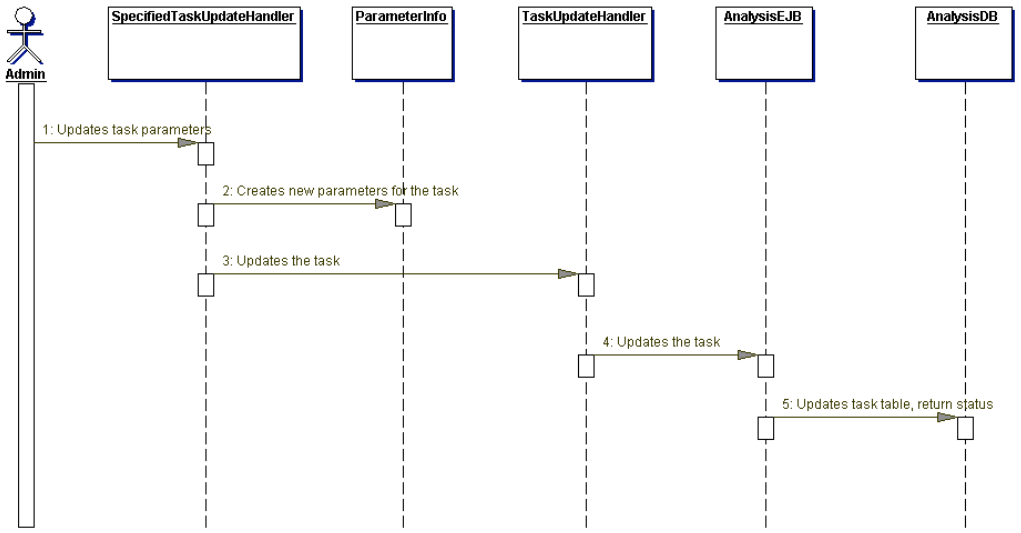
5. Task Management

Add new task:


Admin
TaskLoader
TaskInfo
AddNewTaskHandler
AnalysisEJB
AnalysisDB
1: Creates new task
2: Creates TaskInfo forthe new task
3: Adds new task
4: Adds new task
5: Adds TaskInfo into database


Update Task
ParameterInfo
AnalysisDB
Admin
SpecifiedTaskUpdateHandler
TaskUpdateHandler
AnalysisEJB

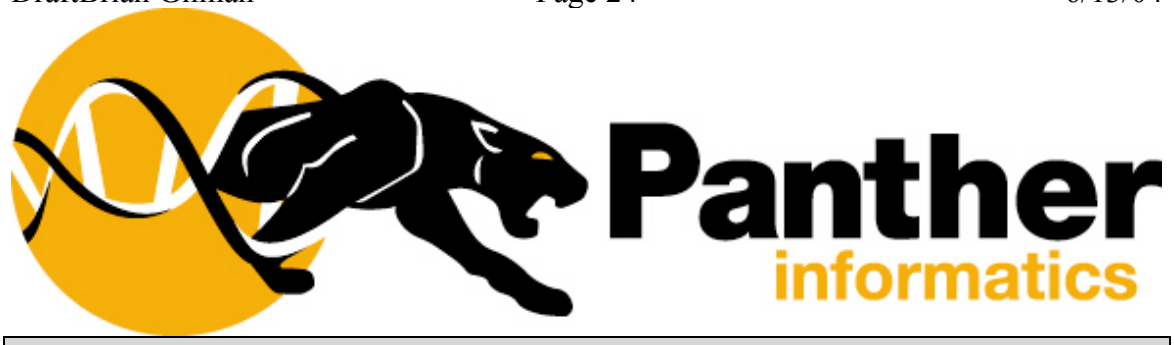
Update task:


Admin
SpecifiedTaskUpdateHandler
ParameterInfo
TaskUpdateHandler
AnalysisEJB
AnalysisDB
1: Updates task parameters
2: Creates new parameters for the task
3: Updates the task
4: Updates the task
5: Updates task table, return status

Analysis Framework Example Code:

This example will be commented in another version of this document. This is taken from one of our test cases found in <omnigene_home>/languages/java/src/tests/webservice. It shows you how to call the OmniGene Analysis Engine from code.

```
import java.util.*;
import java.io.*;

import junit.framework.*;
import org.apache.log4j.Category;

import edu.mit.wi.omnigene.framework.analysis.webservice.client.*;
import edu.mit.wi.omnigene.framework.webservice.WebServiceException;

import edu.mit.wi.omnigene.framework.analysis.*;

public class GetTasksTest extends TestCase
{
    private AnalysisWebServiceProxy proxy = new
AnalysisWebServiceProxy(new
URL("http://somewhere.on.the.internet.com");

    private static Category cat =
Category.getInstance(GetTasksTest.class.getName());


    public GetTasksTest(String name)
    {
            super(name);
    }

    protected void setUp()
    {
    }

    public void testGetTasks()
    {
        cat.info("Testing get tasks.");
        cat.info("WebService URL: " +
tests.webservice.TestConstants.WEBSERVICE_URL );

        try {
            TaskInfo[] ti = proxy.getTasks();

            assertNotNull("No tasks returned", ti);
```




```
            if (ti != null) {
                    for (int x = 0; x < ti.length; x++) {
                        cat.info("\tName:        " + ti[x].getName());
                        cat.info("\tDescription: " +
ti[x].getDescription());
                        cat.info("\tID:          " + ti[x].getID());
                        cat.info("\tParm Info:   " +
ti[x].getParameterInfo());
                        cat.info("\tClass name:  " +
ti[x].getTaskClassName());

                        Map m = ti[x].getTaskInfoAttributes();
                        if (m != null) {
                                TaskInfoAttributes tia = new
TaskInfoAttributes(m);
                            cat.info("\tTask Attr:   " + tia.toString()
+ "\n");
                        }
                        else {
                            cat.info("\tTask Attr:   <none>\n");
                        }
                    }
                }
            }
            catch (WebServiceException wse) {
                cat.error("Failed to get tasks: " + wse.getMessage());
                fail("Failed to get tasks.");
            }
        }

        public void tearDown() {
        }

        public static Test suite()
        {
            return new TestSuite(GetTasksTest.class);
        }

       public static void main(String[] args)
       {
            junit.textui.TestRunner.run(suite());
       }
}
```

Installing And Configuring the OmniGene Analysis Engine Runtime:

We have been hearing a lot of complaints over the last few years about installing and configuring the platform with regard to the Omnigene Analysis Engine and Framework. In response to this we have automated the installation process for the components and runtime systems needed get this subsystem working.

You will need the Ant build tool to get this running. Please see the section entitled:

OmniGene Runtime System

to bootstrap your system with apache's Ant tool.

All you will need to do to install this sub-component it issue the following command:

ant deploy-oae-runtime

This command will download all components needed to use this sub-system, configure the runtime engine and place everything in a convenient place OMNIGENE_HOME/omnigene-x.xx/runtime where x.xx is the build number.

<Warning>
The next step have not been tested on all platforms. Due to threading issues they may fail. This is especially true for Windows users!

After this task has been executed you will need to run one more additional command:

ant execute-runtime-tests

This will finalize the configuration step and install a default "echo task" into the OmniGene Analysis Engine for you to work with. You may see some warnings etc. just ignore them for now, the system is telling you that it has not been initialized.

</Warning>

If the above command fails, quit ant (control-c on unix) and execute the following commands in a terminal window backgrounding each task as you go (in unix execute the bg and control-z command):

ant start-hsqldb &
ant start-jboss &

ant start-tomcat&

Note: No ampersands for the following

ant deploy-echomf-task
ant deploy-analysis-webservice

Finally ant execute-oae-tests

You should see some output and then a

Successful Test at the end.

Congratulations! You've got the OmniGene Analysis Engine working. To keep this going execute the three commands above that have ampersands next to them and you're ready to go.

if you'd like to see this all in action with our viewer application perform the following:

ant jar-omniview
cd OMNIGENE_HOME/build/languages/java/jars
mv omniview.jar ../
edit OMNIGENE_HOME/build/languages/java/resources/omnigene.properties
finding the property analysisServiceURL=@@Analysis_URL@@
replacing the @@xxx@@ with http://localhost:8080/axis/servlet/AxisServlet

Then file up omniview using the following command:

cd OMNIGENE_HOME/build/languages/java/
java –Domnigene.conf=./resources –jar omniview.jar

After the OmniView application has started try clicking on the "analysis" tab and see the list of services that are available. Click on the "echo task" and upload two files to the service. Click on the "History" tab and see if the task's status. From there you can see the results of your task. Double click on the results and you will see the files that you uploaded to the server. This panel and the logic contained within can be found in:

edu.mit.wi.omniview.analysis.*

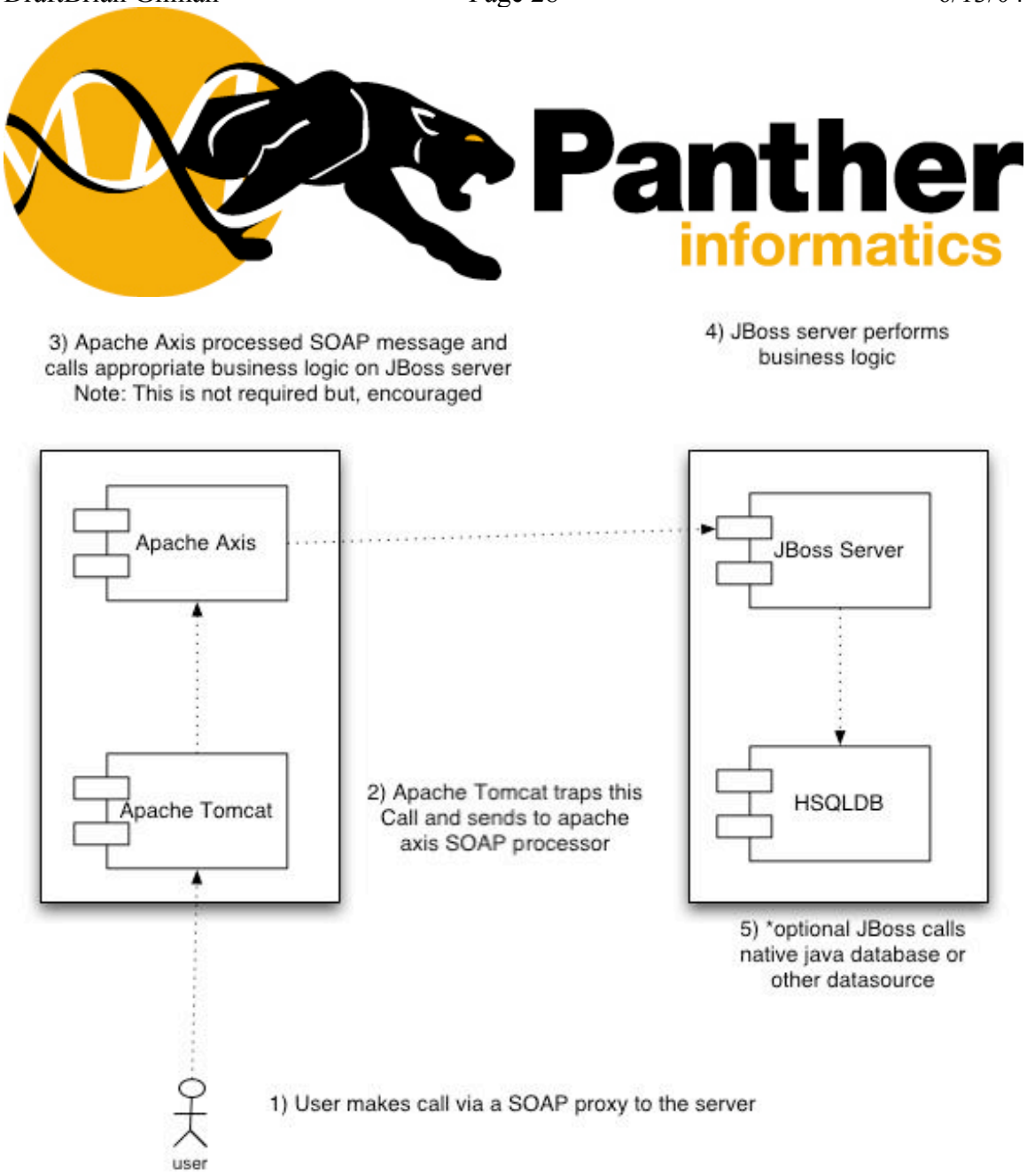
You are free to use these GUI controls for your own application. Have fun!

The Web Service Framework:

Introduction:

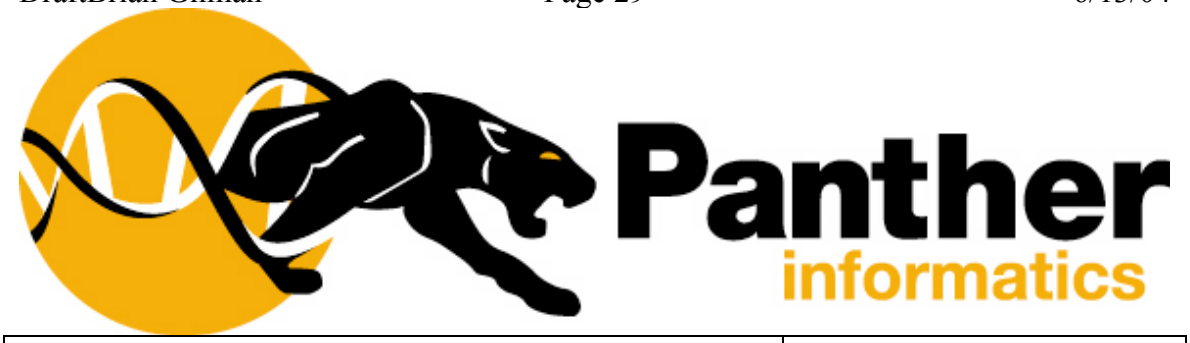
Web services are components written in any language and exposed via a Simple Object Acess Protocol (also known as SOAP). SOAP is a platform and language independent grammer, described in XML, that allows disparate languages and platforms to interact as if they were running in the same process. SOAP allows disparate processes to interact in two separate modes: Remote Procedure Call and Document based mode. RPC mode allows two process running on different machines to interact as if they were running on the same computer. Document based calls allow a process to send a document with arbitrary XML to another process for parsing. The latter type of messaging is useful for dynamic messaging or workflow related tasks. OmniGene utilizes an RPC style of messaging for most if not all of its tasks.

To make the runtime system more concrete a typical sequence of calls to the different pieces of the runtime system are shown in figure X.X below. Please note that processing goes in a counter clockwise fashion starting in the bottom left hand corner. of the diagram.



The OmniGene team has written a small set of interfaces and abstract classes to interface with the Omnigene runtime system. This allows the core development team to maintain control over how processes are controlled, instantiated, deployed, and destroyed. Table X.X describes the **WebService** interface and its methods.

Method Name	Method Description
getWebServiceName	This method returns the name of this webservice as a String
getWebServiceInfo	Returns meta data associated with this web service



	including: textual description of the web service, version of the web service, and the author of the service
getEncodingScheme	returns the enconding scheme used for this web service examples include: HTML/SOAP/BSML etc
setEncodingScheme	used by the runtime engine to set the encoding scheme
getEncondingSchemeVersion/setEncodingSchemeVersion	Used by the runtime engine to get and set the version of the schema that the will be used by omnigene runtime components
ping	This allows the client side interface to test whether this web service is alive or not sends back a text String

Because this interface is quite generic, the omnigene team has written an abstract class called GenericWebService that implements the methods described in Table X.X. This is the abstract class that each and every web service that you write to expose a service in the OmniGene runtime will extend to expose its functionality to the world. This webservice can be found in the package:

edu.mit.wi.omnigene.framework.webservice.GenericWebService and implements the WebService interface.

Note: At some point in the future you will not have to extend this webservice for each and every web service that you write. The Omnigene team is working on allowing you to expose your executables by placing them in a directory for inspection and exposure.

Implementing an OmniGene web service

Developers who are used to working with the Apache Tomcat Servlet engine should have little trouble understanding the implementation strategy of the OmniGene

web service engine. Developers extend the generic web service abstract class and write their custom logic inside their base class. Once they do this they have complete access to all the functionality and context information that Apache Axis, and Apache Tomcat provide. As mentioned above the OmniGene web service engine utilizes these Java Enterprise components to perform most of its work.

The Transform Framework (JAXB):

The Handler Frameworks:

The Security Frameworks:

The OmniDAS Framework:

Introduction:

Before DAS a Bioinformatician wishing to obtain all the annotations available for a particular segment of DNA had to either download the entire GenBank database or troll through all records looking for all BAC clones that existed within the segment he/she needed. This poor soul then had to parse all the data (in different formats) to get out the particular piece of information he/she wanted. The Distributed annotation system has tried to alleviate the screen scraping/database download nightmare that a Bioinformatician has been faced with since the inception of the FASTA format.

The OmniDAS framework is an implementation of the Distributed Annotation System (DAS 1.0[1]) specification originally conceived by Lincoln Stein, Sean Eddy, and Robin Dowel at Cold Spring Harbor Laboratory. The purpose of this specification is to provide researchers and Bioinformaticians a simple and robust solution for obtaining computational or empirically derived sequence and annotations on a particular segment

[1] See http://www.biodas.org

of sequence from a remote data store (such as a database or flat file in fasta or GFF[2] format). This specification calls for a REST[3] like request/response infrastructure where you http-GET a URL and are returned an XML document of annotations that you asked for.

The OmniDAS framework provides a client side API to connect and receive data from a DAS system provided that it follows the DAS 1.0 (1.5 is not implemented) specification. The OmniDAS system is generally used in conjunction with the Transformation framework as these two systems hide all the details of XML parsing and transformation from the programmer. Examples are provided below to get a feeling for how to use these two frameworks.

OmniDAS Design Overview:

OmniDas has been designed to be a standalone framework in the omnigene system. This means that the OmniGene team has not tied this framework to any other third party tools or other OmniGene frameworks.

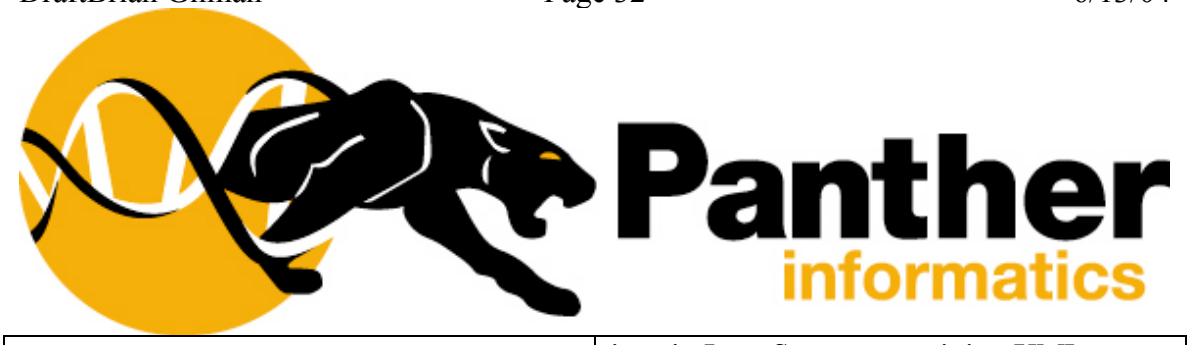
The OmniDAS system was designed to support multiple implementation of the DAS specification. Therefore, we chose to implement DAS using Java Interfaces. Each section of the specification (on the request side) is a java object. The response is returned to the programmer as a java.io.InputStream. This stream can then be tokenized by the Transformation framework or your own XML parser.

Main Framework Classes:

See: http://omnigene.sf.net/docs/images/omnidas.gif

Class Name	Class Description
DASBase	Base Interface that all other interfaces extend (to make everything serializable)
DASRequest	Base Request object that all other request interfaces implement this includes DASDNARequest, DASDSNRequest etc.
DASResponse	Interface that is used to get the response back from the server. This gets you an

[2] http://www.sanger.ac.uk/Software/formats/GFF/GFF_Spec.shtml
[3] http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm



	java.io.InputStream containing XML
DASQuery	Container Class that you use to interact with a particular DAS server this class encapsulates the response and performs your request for you
DASQueryFactory	This class gives you back the correct DASQuery based on the DASRequest you pass to it

Sequence Diagrams:
Coming Soon

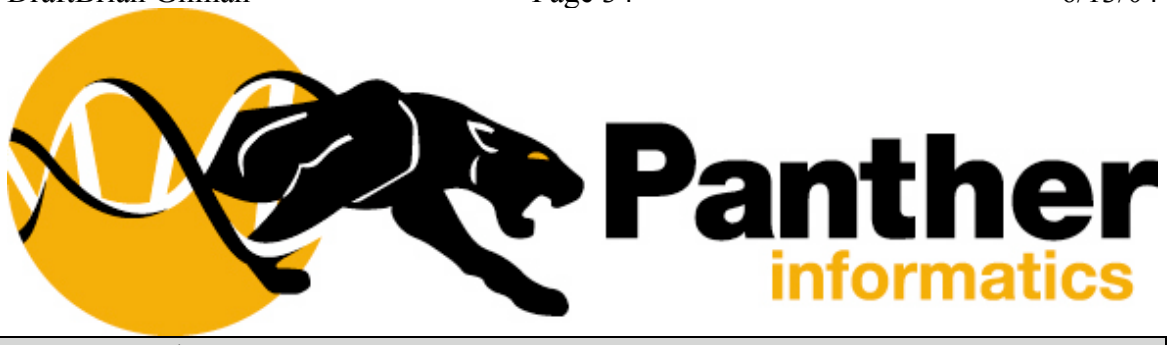
Example Code:

This example shows how to connect to the UCSC DAS server and retrieve XML from the InputStream obtained from the DASResponse.

Example 1:

```
package examples.omnidas;

//Import the omnidas package
import edu.mit.wi.omnigene.framework.omnidas.*;
import edu.mit.wi.omnigene.framework.omnidas.request.*;
import edu.mit.wi.omnigene.framework.omnidas.response.*;

import java.io.*;
import java.net.*;

/**
 * This example will try and highlight some of the
 * functionality of the DAS 1.0 side of the omnidas
 * API.
 *<br>
 *<br>
 * We will:
 *<ul>
 *<li> Show how to connect to a datasource</li>
 *<li> Show how to use the different request objects</li>
 *<li> Show How to use the QueryFactory to get back a DASRsponse</li>
 *<li> Get unparsed XML from the data source</li>
 *@author Brian Gilman
 *@version $Revision: 1.2 $
 */
```

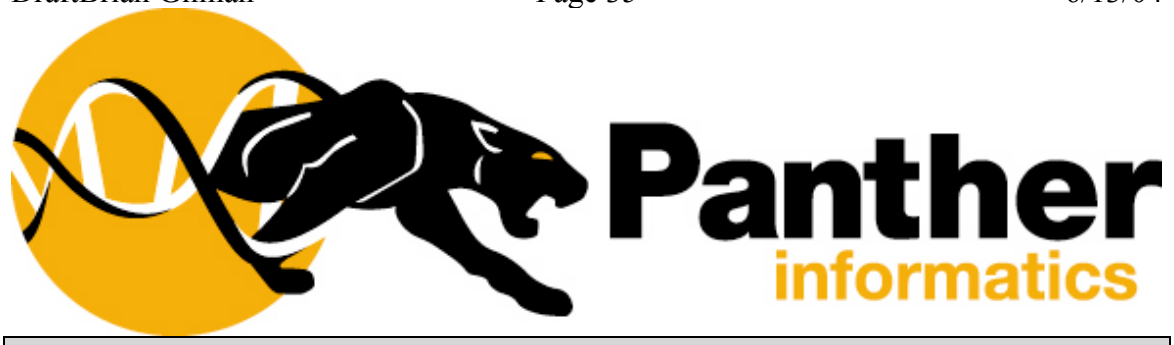

```
public class UCSCDASExample {

    private static final String  UCSC_URL = "http://genome-
test.cse.ucsc.edu/cgi-bin/das";
    //private static final String  UCSC_URL =
"http://www.fruitfly.org/cgi-bin/das";
    private static  DASQueryFactory factory = null;

    public static void main(String[] args){

      try{
          /* We'd like to get all the DSN's for the golden path DAS
service
           * So we need to construct a DASDSNQueryImpl pasing it the
version of the DAS
           * server and the URL for the server
           */

          DASDSNRequestImpl dsnRequest = new DASDSNRequestImpl(1.0f,
new URL(UCSC_URL)); //that's it!

          /*
           * Now we get an instance of the DASQueryFactory
           */

          factory = DASQueryFactory.getInstance();

          /*
           * pass the Request to the factory using the getDASQuery `
            method
           * This will get us the proper instance of a DASQuery
implementation
           * in this case we get back a 1.0 version of the Query object
           */
          DASQuery query = null;

          try{
             query = factory.getDASQuery(dsnRequest);
          }catch(DASException edas){
            edas.printStackTrace();
          }
          /**
           * We can now execute the query against the DAS server
           * the Query class connects to the server automatically for
us
           */

          query.doDASQuery();
```



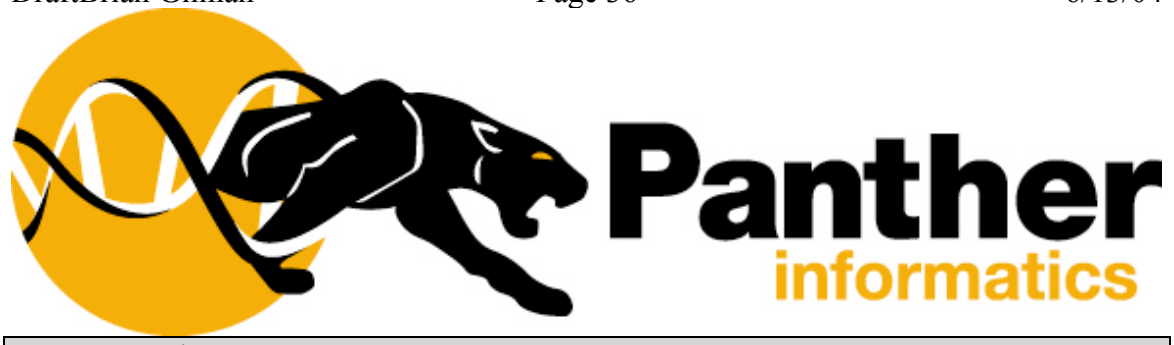

```
            /**
             * now get the DASResponse and print out some XML from the
             * InputStream provided in the response object
             */

            DASResponse resp = query.getDASResponse();

            //print it

            BufferedInputStream in = new
    BufferedInputStream(resp.getResponse());

            int c;

            while((c = in.read()) != -1 ){
              System.out.print((char)c);
            }

          }catch(Exception e){
              e.printStackTrace();
          }
        }
    }
```

Example 2: Getting Annotations From Ensembl And Parsing With JAXB

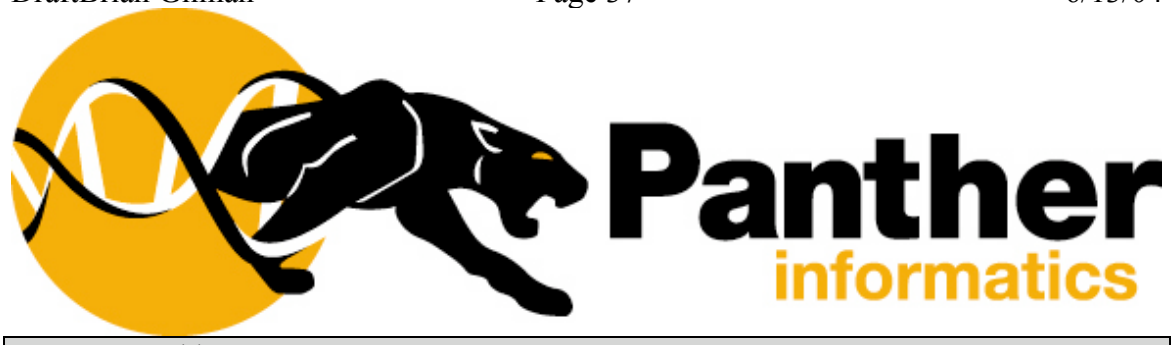
```
package examples.omnidas;

import edu.mit.wi.omnigene.framework.omnidas.*;
import edu.mit.wi.omnigene.framework.omnidas.request.*;
import edu.mit.wi.omnigene.framework.omnidas.response.*;

import edu.mit.wi.omnigene.framework.jaxb.das.dsn.*;
import edu.mit.wi.omnigene.framework.jaxb.das.gff.*;
import edu.mit.wi.omnigene.framework.jaxb.das.entrypoints.*;

//import edu.mit.wi.omnigene.omnibus.*;


import java.net.*;
import java.io.*;
import java.util.*;
import javax.xml.bind.*;
import javax.xml.marshal.*;
/**
 * Simple example which connects to Ensembl asking for
 * DSN and EntryPoints
 *@author Brian Gilman
```



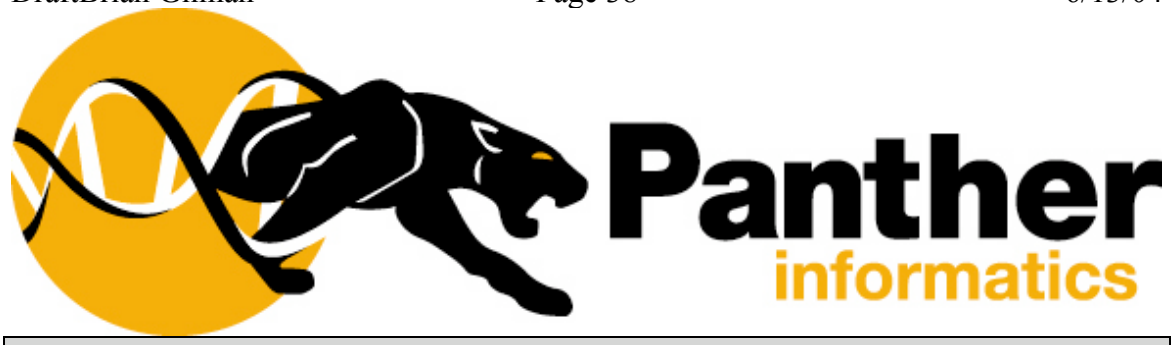

```
 *@version $Revision: 1.3 $
 */

public class EnsemblDASExample{

    private static final String ENS_URL =
"http://servlet.sanger.ac.uk:8080/das";
    //private static final String ENS_URL = "http://genome-
test.cse.ucsc.edu/cgi-bin/das";
    private static DASQueryFactory factory = null;

    public static void main(String[] args){
      EnsemblDASExample ex = new EnsemblDASExample();
      Vector v = ex.getDSN();

      try{
          ex.printIds(v);
          ex.printEntryPoints(ex.getEntryPoints(v));
      }catch(Exception e){
          e.printStackTrace();
      }
    }

    public Vector getDSN(){

      Vector dsns = new Vector();

      try{

          /*
           * Declare an array of DSNImpl to return
           */


          /* We'd like to get all the DSN's for the ensembl DAS service
           * So we need to construct a DASDSNQueryImpl pasing it the
version of the DAS
           * server and the URL for the server
           */

          DASDSNRequestImpl dsnRequest = new DASDSNRequestImpl(1.0f,
new URL(ENS_URL)); //that's it!

          /*
           * Now we get an instance of the DASQueryFactory
           */

          factory = DASQueryFactory.getInstance();
```



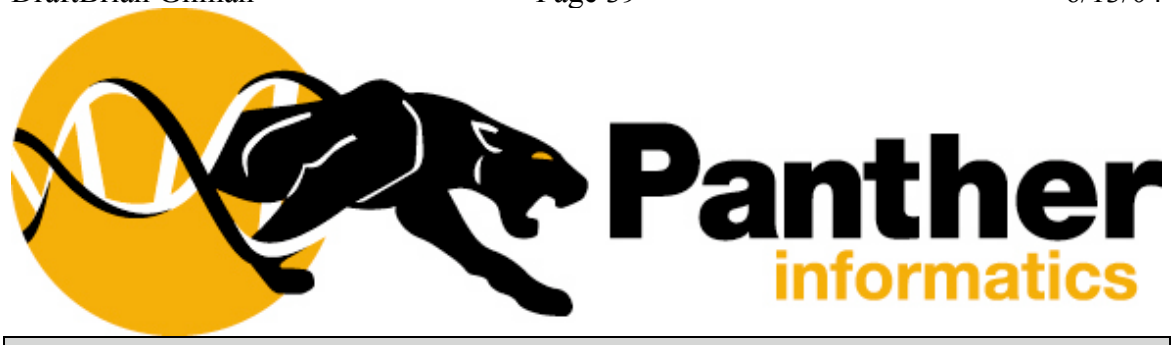

```
          /*
           * pass the Request to the factory using the getDASQuery
method
           * This will get us the proper instance of a DASQuery
implementation
           * in this case we get back a 1.0 version of the Query object
           */

          DASQuery query = null;

          try{
            query = factory.getDASQuery(dsnRequest);
          }catch(DASException edas){
            edas.getURI();
            edas.printStackTrace();
          }
          /**
           * We can now execute the query against the DAS server
           * the Query class connects to the server automatically for
us
           */

          query.doDASQuery();

          /**
           * now get the DASResponse and print out some XML from the
           * InputStream provided in the response object
           */

          DASResponse resp = query.getDASResponse();


          // DASMetaData contains te version of this server
          // as well as the schemaname used to transmit data
          // the version of the schema and the number
          // of charaters transmitted from this request
          DASMetaData dsmd = resp.getDASMetaData();

          System.out.println("DAS Server Version: " +
dsmd.getDASVersion());

          BufferedInputStream in = new
BufferedInputStream(resp.getInputStream());

          // Uncomment below to print XML in BufferedInputStream
          // Notice that this breaks the JAXB below.

          //int c;
          //while((c = in.read()) !=-1 ){
```




```
          //      System.out.print((char)c);
          //}

          //in.close();

          // The DASDSN object knows how to read in our XML stream from
          // the DASResponse object abve and unmarchal it into
          // objects for inspection
          // see www.javasoft.com/xml for a description of JAXB
          DASDSN dasDSN = DASDSN.unmarshal(in);


          in.close(); //always close your streams!

          // Here we use the JAXB method of getting things out of XML
documents
          // This is much like using JDOM where we get out Lists of
objects contained
          // in the XML
          List listOfDSNs = dasDSN.getDSN();

          //Get an Iterator to iterate over the list of DSN's in the
document
          for(ListIterator i = listOfDSNs.listIterator();i.hasNext();){
            // Must cast with fully qualified name because omnidas and
jaxb.das.dsn use the same
            // name ie. DSN
            edu.mit.wi.omnigene.framework.jaxb.das.dsn.DSN dsn =
(edu.mit.wi.omnigene.framework.jaxb.das.dsn.DSN)i.next();


                // Here we instantiate a new DSNImpl from omnidas and
set it's id
                // with the JAXB object's Id
                // DSN's have source objects and sources have id
attributes which
                // we need to instantiate an omnidas DASImpl object
            dsns.add(new DSNImpl(dsn.getSOURCE().getId()));
          }

      }catch(Exception e){
          e.printStackTrace();
      }

      return dsns;

    }

    public void printIds(Vector ids){
```




```
      Enumeration e = ids.elements();

      while(e.hasMoreElements()){
          System.out.println(((DSNImpl)e.nextElement()).getID());
      }
    }

    /**
     * Given a Vector of DSN's print out EntryPoint object id's
     *@param v the vector of DSN's
     *@return a Hashtable where each key is the DSN and each value
     * is a vector of EntryPoints
     */
    public Hashtable getEntryPoints(Vector v) throws IOException,
MalformedURLException, UnmarshalException{
      Hashtable h = new Hashtable();
      Enumeration e = v.elements();
      Vector entries = null;
      BufferedInputStream in;
      DASResponse resp = null;

          while(e.hasMoreElements()){
            DSNImpl dsn = (DSNImpl)e.nextElement();
            DASEntryPointRequest entryPointRequest = new
DASEntryPointRequestImpl(1.0f, new URL(ENS_URL), dsn);
            factory = factory.getInstance();
            DASQuery query = null;
            try{
                query  = factory.getDASQuery(entryPointRequest);
            }catch(DASException edas){
                System.out.println(edas.getURI());
                edas.printStackTrace();
            }

            try{
                query.doDASQuery();
                resp = query.getDASResponse();
            }catch(DASException eDas){
                System.out.println(eDas.getURI());
                System.out.println(eDas.toString());
                continue;
            }

            in = new BufferedInputStream(resp.getResponse());
                //int c;
                //while((c = in.read()) != -1){
                //System.out.print((char)c);
                //}
```




```
            DASEP dasep = DASEP.unmarshal(in);

            in.close();

            ENTRYPOINTS eps = dasep.getENTRYPOINTS();

            List segments = eps.getSEGMENT();
            entries = new Vector();
            for(ListIterator i = segments.listIterator();i.hasNext();){

edu.mit.wi.omnigene.framework.jaxb.das.entrypoints.SEGMENT segment =
(edu.mit.wi.omnigene.framework.jaxb.das.entrypoints.SEGMENT)i.next();
                entries.add(segment.getId());
            }

            h.put(dsn,entries);

          }


          return h;
    }

    public void printEntryPoints(Hashtable entries){

      Enumeration keys = entries.keys();
      Vector v = null;

      while(keys.hasMoreElements()){
          DSNImpl dsn = (DSNImpl)keys.nextElement();
          v = (Vector)entries.get(dsn);
          Enumeration segments = v.elements();
          while(segments.hasMoreElements()){
            System.out.println("DSN: " + dsn.getID() + " has segment: "
+(String)segments.nextElement());
          }
      }
    }

}
```

The Database Frameworks:

The Validator Framework:

The Lookup Framework:

The Bio Framework:

The DBSieve Framework:

Getting To Know OmniGene Services:

The OmniGene Analysis Engine (OAE) Service:

The DAS Service:

The Security Service:

The Single Sign On Service:

The Lookup Service:

The Mailer Service:

The ProjPed Service:

The SNP Service:

The Single Sign On Service (SSO):

Layout Of OmniGene Code Base:

See: http://omnigene.sourceforge.net/Package_Structure_OmniGene2.html

OmniGene Build System:

Note: OMNIGENE_HOME is referred to as the place where you downloaded or cvs checked out omnigene2.

The OmniGene build system utilizes Apache Ant to compile, deploy and test the OmniGene system. OmniGene comes with a unix shell script to bootstrap your OmniGene build (Windows users must download apache ant directly. If you'd like to contribute a batch script to do this we'd be thankful!). To get started with OmniGene all you need to do is execute the following command:

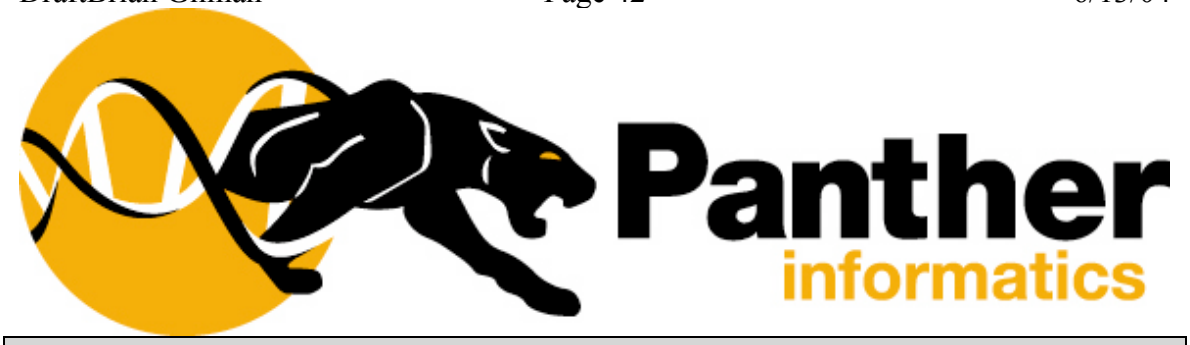
./bootstrap_build.sh

Apache's ant will be downloaded to your computer and installed in OMNIGENE_HOME (wherever you downloaded or cvs checkout(ed) the omnigene2 system. You will need to put the <ANT_HOME>/bin directory in your path to execute ant. After you have completed this task you are ready to build the OmniGene system. As a test try executing the following command in your OMNIGENE_HOME directory:

ant

You should see the following output:

```
antenv:
    [echo] ---------------------------------------------------------------
    [echo]         Build environment for OmniGene 1.1.3 [2003]
    [echo] ---------------------------------------------------------------
    [echo] Ant version : Apache Ant version 1.5.1 compiled on October 2 2002
    [echo] Java version: 1.4.1_01
    [echo] ---------------------------------------------------------------
    [echo] Javac target     : 1.3
    [echo] Javac debug      : on
    [echo] Javac deprecation: false
    [echo] ---------------------------------------------------------------

targets:
    [echo] ------------------------------------------------------------------------------
    [echo]                         Targets
    [echo] ------------------------------------------------------------------------------
    [echo] compile-omnigene            : compiles all omnigene framework source code
    [echo] compile-omniview     : compiles all omniview source code
    [echo] compile-examples     : compiles all example source code
    [echo] compile-tests               : compiles all test source code
    [echo] compile-all                 : compiles everything
    [echo] jar-omnigene            : jars omnigene core (see docs for details)
    [echo] jar-omniview                : make self executing jar filefor viewer
    [echo] jar-lookup-service-mbean    : see docs for details
    [echo] jar-lookup-service          : see docs for details
```




```
     [echo] create-lookup-jars          : creates two targets above
     [echo] create-analysis-jars        : creates analysis service jar files
     [echo] jar-snp-service             : creates snp service jars
     [echo] jar-sso-service             : creates single sign on jars
     [echo] jar-projped-service         : creates projped ejbs and webservice
     [echo] jar-mailier-service         : creates mail service
     [echo] jar-omnidas                 : creates omnidas framework
     [echo] jar-security            : creates security ejb
     [echo] jar-authenticator           : creates authentication service
     [echo] ear-application             : creates full omnigene runtime as ear application
     [echo] jar-omnigene-examples       : creates examples
     [echo] create-all-jars             : creates all jar targets
     [echo] dist                    : creates the complete binary distribution
     [echo] srcdist                  : creates the complete source distribution
     [echo] javadocs                   : creates the javadocs
     [echo] clean                    : clean up files and directories
     [echo] targets                  : displays list of available targets
     [echo] ---------------------------------------------------------------------------------

BUILD SUCCESSFUL
Total time: 4 seconds
```

If you do not see the output shown above please make sure that apache-ant is in you path and try reading the apache-ant documentation online.

If you are successful, compiling OmniGene is simple, execute the following command:

ant jar-omnigene

The output of this command will be placed in:

OMNIGENE_HOME/build/languages/java/jars/omnigene.jar

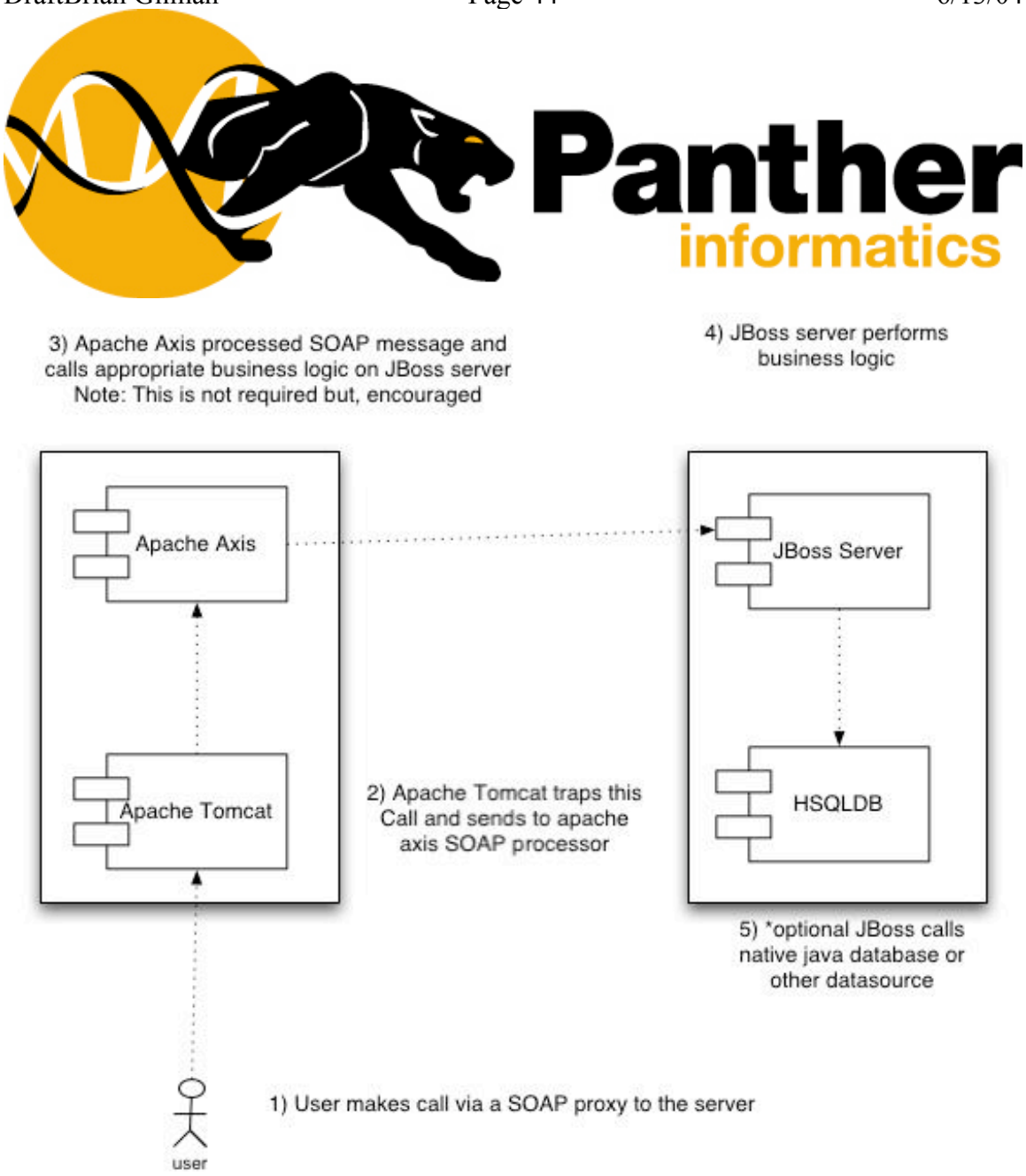
This command builds the entire platform and is a good way to test whether your system is configured properly.

OmniGene Runtime System:

Introduction:

The OmniGene runtime system is dependent on three main components: Apache's Tomcat, JBoss, Apache Axis and HSQL (embedded java database). These runtime components are used in conjunction to provide the following functionality:

Runtime Component	Functionality
Jakarta-Tomcat	Provides a runtime environment for the Apache Axis System. it is used as a servlet engine. However, any servlet engine that can host apache axis (BEA Weblogic/Jrun/Resin can be used here
JBoss	Acts as both an Mbean server (Analysis Engine, Lookup Service) and an Enterprise Java Bean runtime system. This system is used to access local databases (proj/ped service) as well as embedded databases (HSQL)
Apache Axis	Apache Axis is used as the default SOAP system in the omnigene frameworks. All SOAP services are deployed inside this system and are exposed through Axis.
HSQLDB	This is used as our default embedded database as it is platform independent and fairly fast. The OmniGene Analysis Engine uses this database. Other components may use this in the future



Testing Provisions:
Running OmniGene Tests:
Running The OmniGene Examples:

Appendix

Recommended Coding Practice
Using CVS


Panther
informatics