2007

A distributed system for integrating and sharing biology data and tools

Mohammed Alabsi
Iowa State University

A distributed system for integrating and sharing biology data and tools

by

Mohammed Alabsi

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Science

Program of Study Committee:
Leslie Miller, Major Professor
Wallapak Tavanapong
Drena Dobbs

Iowa State University

Ames, Iowa

2007

To my parents

TABLE OF CONTENTS

ABSTRACT

During the last few years, the number and size of biological databases have increased dramatically. Existence of these databases has been significant to recent biological research, providing biologist with an increasingly large and valuable set of data. Still, current biological databases have some shortcomings. A single data source does not usually have all the data a biologist needs. Also, biological databases are normally subject specific, containing data about a certain biological discipline. Hence, providing biologists with a single source of data gathered from multiple sources and covering many subjects would provide them with the breadth and depth of data needed for many biological studies. Such source should also include the needed tools to conduct necessary analytical studies on the integrated data.

A system model is introduced that gathers integrated data from multiple diverse biological databases, enables biology labs to share their data with each other, and provides support for analytical tools relevant to biological research of users. External tools can also be attached to the platform using the notion of plug-ins. The design of the system is provided and the implementation details are explained.

1. INTRODUCTION

The usage of technology and prediction programs as a cheap and fast way to generate biology data has dramatically increased the amount of biological data available. Research institutes, educational institutes and companies have created their own databases containing the data they discover. As a result, the number of publicly available databases has been growing dramatically. Galperin declared the number of publicly available databases to be 968 in 2007, compared to 858 in 2006 and 719 in 2005 [8, 9]. The existence of these databases has been significant to recent biological research, providing biologists with increasingly large and valuable data. Still, current biological databases experience some shortcomings: typically, a single data source does not have all the data a biologist needs. Biological databases are normally subject-specific. Since many biological experiments require data from different domains, obtaining access to data distributed across multiple data sources is essential.

Several difficulties are faced when integrating data of different origins. The data may have different formats, and are usually stored in heterogeneous systems. Along with syntactic differences, semantic gaps are normally present. A word or phrase in one source may have a different meaning in another. Only by overcoming such challenges, can the data be properly integrated. To bridge these problems, we would like to expand the traditional notion of "biological data warehouse", which focuses on solving the problem of biological data integration, to handle the integration analytical and visualization biological tools as well. Furthermore, a data warehouse could be thought of as a "user community" where users can add their own data, and share it with others; as well as having the ability to plug in their own tools to the platform. Hence, giving users the power and flexibility needed to customize the platform the way it suits their needs.

In this paper, a data warehouse system for integrating and sharing biological data and tools is designed and implemented. The warehouse system provides users with the ability to add databases or tools to search data gathered from public sources or shared by others, and access to a number of biology tools available on the server. Web services are used to query and

retrieve data from public data sources, and a Hibernate-based object-relational database is used to store the complex biological entities cached from the available data sources.

A survey of the recent work in the area of biological data integration is described in Chapter 2. The general concepts of web services and their architectures are surveyed in Chapter 3. Object-relational DBMSs and the basics of Hibernate are introduced in Chapter 4. The design goals for the system are discussed in Chapter 5. System usage is described in Chapter 6. Finally, the conclusions about this work are drawn and future improvements related to this work are given in Chapter 7.

2. BACKGROUND

The issue of biological data integration has been addressed since late nineties; as solutions and systems, following different design patterns and goals, have been developed. The first generation of systems was navigational-based, where textual data is indexed and navigated every time a query is posted. The most successful was the Sequence Retrieval System (SRS), which is developed with the assumption that the data sources are text-based. SRS parses data files and as they are scanned, it creates indexes used to speed up the search process. No ontology or dictionary is used to find semantically equivalent entries, rather links are established between entries that have similar words. Still, SRS has its short comings: scanning all the data files, which can be in the order of terabytes, is very expensive; also, there is the need to re-scan the data files every time there is an update; linking entries that have similar words does not necessarily provide an efficient semantic integration mechanism; and finally, SRS does not address biological tools integration [11].

Later systems were either mediator-based, systems with un-materialized databases where data sources are queried on demand, or data warehouse systems, where data was gathered from the data sources and stored in the database. The first of such systems was K2/Kleisli, which is a mediator-based system where the mediator communicates with the data sources through wrappers. User queries, which are expressed in CPL (later replaced with OQL in K2), where passed to the mediator, which in turn generates a query for each data source. Wrappers, receiving the CPL query, translate it to a source-based query and return the results back to the mediator. Although this system was an improvement over SRS, in the sense the data sources' data can be in any format, it lacked any solution to impose semantic data integration. It also did not address tools integration as well [6, 7].

Another popular mediator system is TAMBIS (Transparent Access to Multiple Bioinformatics Information Sources). TAMBIS is divided into three major components: conceptual model, a dictionary of elementary terms that could be used to for complex terms and is organized in a hierarchy based on "is_a" relationships; source model, which provides description of the underlying sources and mappings between the sources and terms from the

conceptual model; and a user interface which allows users to formulate queries. The interface also includes a navigation tool, which lets the user navigate terms in the conceptual mode, and once a term of interest is found, the term can be used to formulate a query. TAMBIS is better than K2/Kleisli in the sense it addressed semantic integration, but it still does not handle tools integration [4].

A popular data warehouse system is BioWarehouse, which physically parses and loads data from the integrated data sources into its database. Such approach has many advantages: system performance is generally much better than in an on-demand environment, as communication with external data sources is eliminated; system reliability should be higher since the number of dependencies (data sources) is much less; and more query optimization techniques could be implemented. Still, this approach suffers from the high cost of building and maintaining such system along with the fact that such data warehouses are out of date every time the data sources are updated. Hence, they need to be updated continuously. Although BioWarehouse uses Gene Ontology to find semantically related entities, it does not address tools integration [14].

Middleware integration systems are a category of systems that are directed towards bioinformatics developers, with interest to develop integration systems, rather than biologists. IBM's Discovery Link was the first successful middleware integration system. It integrated data sources by generating a wrapper for each source, which translates the data source model and describe the data available in the underlying sources. Using the information provided by the wrappers, the query processor breaks the query into portions that be handled by each source. Applications would submit queries to DiscoveryLink, in SQL format, over the global schema without knowing the schema of the data sources [10]. With the recent popularity of web services, many biological institutes have developed web services to provide programmatic access to their data and tools. The European Bioinformatics Institute (EBI) is one of the first institutes to adopt web services. Their web services provide biological data (including PDB, EMBL, Interpro, UniprotKB and Medline), and tools such as Fasta and Blast [15]. Both DiscoveryLink and EBI's web services require a user interface in order to be used by biologists.

Hence, although issue of physical data integration has been addressed by all of the above systems, semantic integration was left out by some of them. Many systems did not handle tools integration, and none allow users to add their own data (and share it with other users) or attach their tools to the system. As a result, we believe that the system presented in this work, is the only current solution that encloses all of these features.

3. WEB SERVICES

The W3C defines a web service as a “software system designed to support interoperable machine to machine interaction over a network” [13]. They are software applications that are self-contained and self-describing. They can be published, discovered and used by other applications. Web services can perform simple tasks, like currency conversion and weather reports, or complex tasks like business processes. Web services receive requests over open protocols (HTTP and HTTPS are most commonly used), performs the necessary operations and returns its response to the requesting machine. Both the request and response are formatted in the eXtensible Markup Language (XML), whose primary usage is to facilitate the exchange and sharing of data between different systems by describing data in a tree-based structure that is readable by both humans and machines.

Many approaches to distributed computing have been explored and used in recent years, but none has experienced the fast growth that web services have recently seen. So what are the reasons behind that? We can summarize the main advantages of web services as follows:

1. Reusability: There are situations where the same feature(s) are needed by a number of applications. So instead of each of these applications building its own from scratch, they can all request these features from a single web service. Hence, saving application developers a lot of time and eliminating unnecessary redundancy.
2. Data Exchange: As web services are based on XML, they are both programming languages and platforms independent. Hence, a web service implemented in C# on a windows machine can be used buy a Java client application on a Linux machine and vice-versa.
3. Firewall Friendly: Since web services are implemented over open protocols, such as HTTP/HTTPS, they are not blocked by firewalls. This is major advantage over other software-to-software communication protocols such as mobile agents or sockets, where uncommonly used ports must be opened in order for communication to take place.

3.1 Specifications

Web services are composed of core specifications that are usually used by all web services and other additional specifications that are used when the technology of choice dedicates. Section 3.1 describes web services' core specifications, while Section 3.2 explains the process of invoking web services.

3.1.1 SOAP

SOAP stands for Simple Object Access Protocol. It is a communication protocol used to exchange XML-based messages between software applications over a network connection, most commonly HTTP. SOAP was first designed by Dave Winder, Don Box, Bob Atkinson, and Mohsen Al-Ghosein in 1998 as is an object-access protocol. Currently, SOAP is a W3C standard and provides the foundation for Web Services communication.

Both SMTP and HTTP protocols can be used by SOAP to transport messages between applications, although HTTP is used more widely. SOAP may also use HTTPS to provide the necessary encryption for message passing between software applications. Because SOAP uses such internet protocols for communication, its messages are not blocked by firewalls. SOAP formats its messages using XML, which is widely used to exchange and share data. Still, SOAP has some limitations. Since it uses XML, it can be slower than other middleware technologies like CORBA. Also, most of the SOAP implementations have limitations on the amount of data sent by each message.

A basic SOAP message is constructed from the required elements of Envelope and Body. It can also have other optional elements such as Header and Fault (Figure 3.1).

```
<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">

<soap:Header>
  ...
  ...
</soap:Header>

<soap:Body>
  ...
  ...
  <soap:Fault>
    ...
    ...
  </soap:Fault>
</soap:Body>

</soap:Envelope>
```

Figure 3.1: SOAP message template [17].

- Envelope: Identifies the document as a SOAP message. The namespace “xmlns:soap” should always be used in Header, with the value of “http://www.w3.org/2001/12/soap-envelop”. Otherwise, an error will be shown and the message will be disposed. An “encodingStyle” attribute is used to describe the data types used in an element. This attribute can be used by an element in the SOAP message, and it applies to children of the element it is described in.
- Body: Contains the SOAP message intended to be received by the proper endpoint. Such a message can be a “request” message or a “response” message. For example, in Figure 3.2a, the message is a request for “GetWeatherReport” operation for zip code 50014. Figure 3.2b, shows the associated response message where the temperature in both Celsius and Fahrenheit is described.
- Header: This optional portion of the message contains application-specific information. If present, it must be the first child of the Envelope element. SOAP defines three default attributes in the Header element:
    - *Actor:* Specifies the particular machine or end-point for which this header is intended through the message path.

- *mustUnderstand:* Indicates whether it is optional or mandatory for the endpoint to process the header.
- *encodingStyle:* As described in Header element.

- Fault: If an error message is carried in the SOAP message, it will be described in the Fault element. It mentions the error code, who/what caused the error, an explanation regarding the error, and other details if necessary [17].

```
POST /InStock HTTP/1.1
Host: www.example.org
Content-Type: application/soap+xml; charset=utf-8
Content-Length: 250

<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
  <soap:Body xmlns:m="http://www.example.org/weather">
    <m:GetWeatherReport>
                  <m:ZipCode>50014</m:ZipCode>
    </m:GetWeatherReport>
  </soap:Body>
</soap:Envelope>
```

Figure 3.2a: SOAP request message asking for weather report.

```
HTTP/1.1 200 OK
Content-Type: application/soap+xml; charset=utf-8
Content-Length: 250
<?xml version="1.0"?>
                          <soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
  <soap:Body xmlns:m="http://www.example.org/weather">
    <m:GetWeatherReportResponse>
      <m:F>80</m:F>
      <m:C>27</m:C>
    </m:GetWeatherReportResponse>
  </soap:Body>
```

Figure 3.2b: SOAP response message with weather information described.

Figure 3.2a shows that the HTTP protocol is used to communicate the request message, and POST is used as the request operation. The host name, content type and content length are

also specified. The HTTP response code is mentioned in the top of the response message; in this case it is 200, resembling status “success”.

3.1.2 WSDL

The Web Services Description Language, or WSDL, is an XML-based language that is used to describe web services and the features they provide. It is usually published by the service developer to describe the functions provided by the service, the inputs required for each function and the output they provide, and the complex data types the service uses. Complex types are data structures that are supported by WSDL by default; hence it is up to the developer to specify the serialization and de-serialization of such objects. Since WSDL is XML-based, it not only works as the basis of client server communication, but it can also provide the client developer with the necessary documentation. The current version of WSDL is 2.0, which supports binding all HTTP request methods, not only POST and GET, as in Version 1.1.

```
- <complexType name="PdbProtein">
  - <complexContent>
    - <extension base="tns1:Protein">
      - <sequence>
          <element name="author" nillable="true" type="xsd:string"/>
          <element name="caveat" nillable="true" type="xsd:string"/>
          <element name="classification" nillable="true" type="xsd:string"/>
          <element name="compound" nillable="true" type="xsd:string"/>
          <element name="depDate" nillable="true" type="xsd:dateTime"/>
          <element name="expData" nillable="true" type="xsd:string"/>
          <element name="journalArray" nillable="true" type="tns1:ArrayOfPdbJournal"/>
          <element name="keywordArray" nillable="true" type="tns1:ArrayOfPdbKeyword"/>
          <element name="obsDate" nillable="true" type="xsd:dateTime"/>
          <element name="replaceEntArray" nillable="true" type="tns1:ArrayOfReplaceEnt"/>
          <element name="source" nillable="true" type="xsd:string"/>
          <element name="title" nillable="true" type="xsd:string"/>
        </sequence>
      </extension>
    </complexContent>
  </complexType>
```

Figure 3.3a: PdbProtein, a complex data type described in UserQuery’s WSDL file.

```
- <wsdl:message name="getDomainAttrNamesResponse">
    <wsdl:part name="getDomainAttrNamesReturn" type="impl:ArrayOf_xsd_string"/>
  </wsdl:message>
- <wsdl:message name="getThreadStateMapResponse">
    <wsdl:part name="getThreadStateMapReturn" type="apachesoap:Map"/>
  </wsdl:message>
- <wsdl:message name="getIpHeadersResponse">
    <wsdl:part name="getIpHeadersReturn" type="impl:ArrayOf_xsd_string"/>
  </wsdl:message>
- <wsdl:message name="createAccountResponse">
    <wsdl:part name="createAccountReturn" type="tns1:UserAccount"/>
  </wsdl:message>
- <wsdl:message name="getCompleteColumnListRequest">
    <wsdl:part name="domain" type="xsd:string"/>
  </wsdl:message>
- <wsdl:message name="removeTableResponse">
    <wsdl:part name="removeTableReturn" type="tns1:UserAccount"/>
  </wsdl:message>
  <wsdl:message name="getPdbHeadersRequest"> </wsdl:message>
- <wsdl:message name="getUniprotHeadersResponse">
    <wsdl:part name="getUniprotHeadersReturn" type="impl:ArrayOf_xsd_string"/>
  </wsdl:message>
- <wsdl:message name="dbExistsResponse">
    <wsdl:part name="dbExistsReturn" type="xsd:boolean"/>
  </wsdl:message>
```

Figure 3.3b: Some of the request and response operations from UserQuery's WSDL file.

Figure 3.3a shows an example of a complex data type used by UserQuery, the main Web Service used in this integrated platform. It first describes the name of the object, PdbProtein, and that it extends the class Protein. Then it shows the group of attributes associated with this object. Please note that the attributes of Protein are not show here. Figure 3.3b shows some of the request and response operations of UserQuery. If the operation is a request, the needed parameter will be described in the attribute "wsdl:part"; while if the operation is a response one, "wsdl:part" will describe the return type. For example, "dbExistsResponse" returns a Boolean value, while the "getCompleteColumnListRequest" requires an argument of type string.

3.1.3 UDDI

The Universal Description Discovery and Integration, UDDI, is an XML-based repository for storing information about web services, enabling everybody to publish their services and discover other services of interest. UDDI registrations consist of three components:

- *White Pages:* contains address and contact information.
- *Yellow Pages:* describes the category of the Web Service based on standard taxonomies.
- *Green Pages:* Technical information about the service published.

The UDDI data model consists of four components:

- *businessEntity:* the top level entity, describing the entity for which the information is registered.
- *businessService:* describes services that contain bindingTemplates.
- *bindingTemplate:* Information needed to call a given service.
- *tModel:* a unique identifier for a given service [16].

3.2 Web Services Invocation

After looking at the necessary components of a web service, we go through an example of web services usage, explaining each of its steps (Figure 3.4). For this example, a client is interested in retrieving weather reports from an available web service:

1. Since the client does not know where and which service to query, it asks a web service locator (or UDDI) for web services that provide weather reports.
2. The service location returns the URL, or location, of the matching web service, if any.
3. Although the client does know the web service location, it does not know the exact function to invoke, what its parameters are, and what is the return type. Hence, the client asks for the description of the web service.
4. The web service shows the client the WSDL documentation that describes the operations available.

5. The client uses the WSDL documentation to lookup the operation to be called and what parameters are needed. In this case, the required parameter is the zip code of the desired city.

Please note that steps 1 and 2 are not needed if the service location is already known.

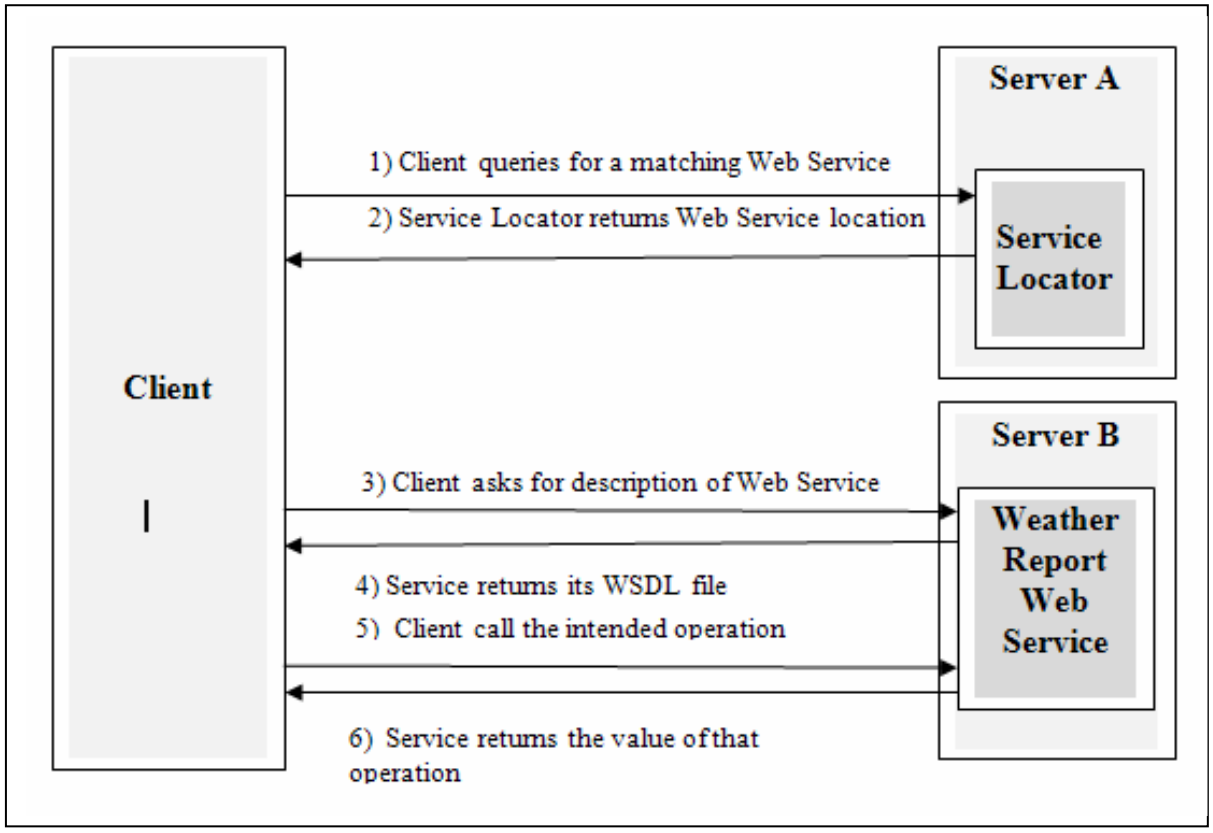

Figure 3.4: Sequence of invoking web services.

4. HIBERNATE

Hibernate is an open-source, object-relational mapping (ORM) tool for Java applications. It provides the framework to easily map Java object-oriented models to relational databases (ex. MySQL). Hibernate relieves the developer from significant amount of common data persistence-related programming tasks. It provides data query and retrieval facilities and significantly reduces the time spent on manually handling SQL and JDBC calls. Hence, when using Hibernate, the developer does not worry about how the objects are stored in the database as records or how data is retrieved as objects. Hibernate is free and is distributed under the LGPL license (http://hibernate.org).

4.1 Architecture

Any application that uses Hibernate is composed of three major components (Figure 4.1):

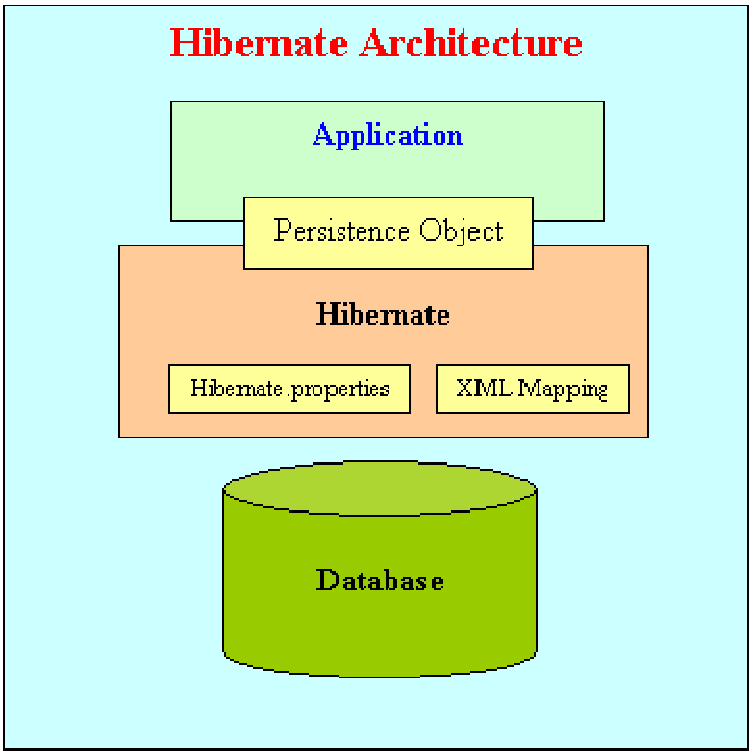

Figure 4.1: High-level model of Hibernate architecture [12].

1) The application level, the Java code, is written to access Hibernate operations and retrieve data objects. Each Java data object, stored or to be stored into the database,

must be represented by a Java bean class where each attribute is a serializable type (no primitives) and each has a public getter and setter.

2) The Hibernate level: Works as an intermediate level between the Java code and the relational database. It receives the requests and operations from the user, either as Hibernate methods or HQL queries, and converts them into SQL before applying them to the database. Along with the Hibernate jars, this level also contains:

    a. Configuration File:

    The configuration file is an XML-based file containing information about the underlying database upon which Hibernate will operate. It describes the driver by which Hibernate should connect to the database, the database name and location, the username and password to access the database. It also describes some of the Hibernate options such as the number of JDBC connections, SQL dialect used, Hibernate session management option, second-level caching option, whether the SQL statements executed by Hibernate should be printed out or not, and finally whether the tables specified in the Hibernate mappings should be generated or not. The last part of the configuration file contains links to all of the mapping files to be used. A sample configuration file is illustrated in Figure 4.2.

    XML has been chosen because it is a standard way to represent data, it is easily parsed and a number of good parsers (like DOM and SAX) are already available, and because it can be easily read by both machines and humans. The configuration file is read once by Hibernate at run-time, and its information is used to establish a connection to the database, and build the mapping between the Java files using in the application and the corresponding database tables. To perform the later task, the names of the mapping files specified in the configuration file must represent valid mapping files. Mapping files are explained next.

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
    <session-factory>

    <!-- Database connection settings -->
        <property name="connection.driver_class">com.mysql.jdbc.Driver</property>
        <property name="connection.url">jdbc:mysql://localhost/bioWarehouse</property>
        <property name="connection.username">xxxx</property>
        <property name="connection.password">xxxx</property>

    <!-- JDBC connection pool (use the built-in) -->
        <property name="connection.pool_size">1</property>

    <!-- SQL dialect -->
        <property name="dialect">org.hibernate.dialect.MySQLDialect</property>

    <!-- Enable Hibernate's automatic session context management -->
        <property name="current_session_context_class">thread</property>

    <!-- Disable the second-level cache -->
        <property name="cache.provider_class">org.hibernate.cache.NoCacheProvider</property>

    <!-- Echo all executed SQL to stdout -->
        <property name="show_sql">false</property>

    <!-- Drop and re-create the database schema on startup -->
     <!--<property name="hbm2ddl.auto">create</property>-->
        <mapping resource="edu/iastate/cs/database/entitiesConfig/go/GoOntology.hbm.xml"/>
        <mapping resource="edu/iastate/cs/database/entitiesConfig/go/GoRelationship.hbm.xml"/>
        <mapping resource="edu/iastate/cs/database/entitiesConfig/go/Synonym.hbm.xml"/>
        ....
        <mapping resource="edu/iastate/cs/database/entitiesConfig/pdb/PdbKeyword.hbm.xml"/>
        <mapping resource="edu/iastate/cs/database/entitiesConfig/pdb/PdbJournal.hbm.xml"/>
    </session-factory>
</hibernate-configuration>
```

Figure 4.2: A sample Hibernate configuration file.

b. Mapping File:

A mapping file is used to map a Java class, with all of its attributes, to a relational database table. It works by first mapping the Java class to a specific table. It then maps each attribute in the Java class to a column in the table. Note that all Java class attributes should be included in the mapping, but it is not necessary that each table column is mapped.

Figure 4.3 provides an example of how a data object is mapped using Hibernate. In this example, “Synonym” is a Java class with three attributes: id (Integer), synonym (String) and type (String) as shown in Figure 4.3a. In the mapping file (4.3b), the first thing specified is that table “synonym” is mapped to the Java class “Synonym”. Afterwards, each attribute (or property) in the

class is mapped to a column in the table. The tag “<generator class =”native”/> in the id column means that it is an auto-increment valued column.

```
package edu.iastate.cs.database.entities.go;

 * @author Mohammed Alabsi <alabsi@cs.iastate.edu>
public class Synonym {

    private Integer id;

    private String synonym;

    private String type;

    public String getType() {
        return type;
    }

    public void setType(String type) {
        this.type = type;
    }

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getSynonym() {
        return synonym;
    }

    public void setSynonym(String synonym) {
        this.synonym = synonym;
    }
}
```

Figure 4.3a: Synonym as a Java class.

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
'http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
    <class name="edu.iastate.cs.database.entities.go.Synonym" table="Synonym">
        <id name="id" column="id">
            <generator class="native"/>
        </id>

        <property name="synonym">
            <column name="synonym" sql-type="text"/>
        </property>
        <property name="type"/>

    </class>
</hibernate-mapping>
```

Figure 4.3b: Hibernate mapping file for the class Synonym.

3) The Database level, where the actual data is being stored and retrieved. Hibernate is relatively DBMS independent, so it may run over MySQL, Oracle, DB2, etc … as long as the proper JDBC driver is specified.

4.2 Sessions, Transactions, and Object States

Within the application, to start using Hibernate functions, a Hibernate session must first be created. A Hibernate session is a single-threaded object that represents communication between the application and the database. It wraps around JDBC connections and behaves as a transaction factory, where a transaction is a set of instructions that must be performed as a unit of work. All Hibernate operations must exist within the context of a session. A session is obtained from a session factory object, which is a cache of compiled mappings that are specified in the mapping files.

Within a session, before any Hibernate function is invoked, a transaction object must be created. A transaction specifies is important when a group of Hibernate operations are related to each other and the failure of one would result in the failure of the entire sequence of operations. In such cases, a rollback can be invoked, and any changes to the data would be undone. Once a transaction is created, objects can be saved to the database using “save”, updated using “update”, removed from the database using “delete”, or retrieved using “get”.

In all of these operations, the object id (table's primary key) and the object type are used as an identifier of the data object. HQL (Hibernate Query Language) or SQL queries can also be invoked when what is needed is more than storing, deleting or update an object.

Within the application, an object may be in any of the following states:

- *Transient:* An object that has never been associated with the persistent context. In other words, it is an object that never been stored in the database (and does not have a primary key). A transient object can be changed to persistent by saving it using the "save" operator.
- *Persistent:* An object that is associated with the database. It is an object that has an id (or primary key) and has a corresponding row in the database. Only for persistent objects can an application retrieve related data (using foreign keys).
- *Detached:* An object that was once associated to the database, but it's Hibernate session (where the association took place) was lost or disconnected. For this object to be persistent again, it needs to be reloaded in a new session [12].

In the next chapter, the warehouse system design and goals are examined.

5. SYSTEM DESIGN AND GOALS

In this chapter, a complete description of the system architecture and design will be explained, while describing how the components behave and interact with each other (Figure 5.1).

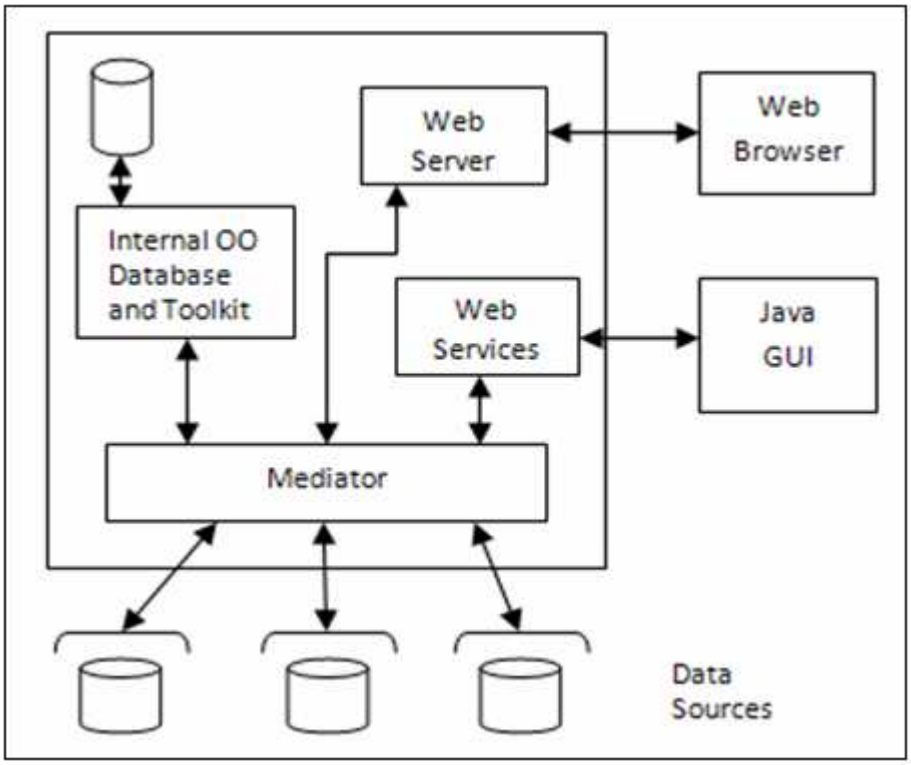

Figure 5.1: Model of the system architecture.

5.1 Graphical User Interface (GUI)

The most visible part of the system, and the component the user will directly interact with, is the graphical user interface (GUI). The GUI, shown in Figure 5.2, is a light-weight Java-based interface that allows the user to interact with the server-side of the system. In other words, the GUI receives users' requests and passes them to the server, and then gets the results from the server and shows them to the user. Through the GUI, users can send search queries (for DNA, RNA, or Proteins), run server-side tools in data, or plug in their own tools to the GUI and run them on the data retrieved.

The GUI communicates to the server through SOAP web services, which provides the flexibility of changing some functionality on the server, or modifying the data sources integrated, without the need to update the user's GUI. It also ensures that users will only need to download a small, light-weight package of software to their desktops rather than downloading the whole system. There will be future work to build an AJAX web-based interface that will include most of the features in the desktop GUI.

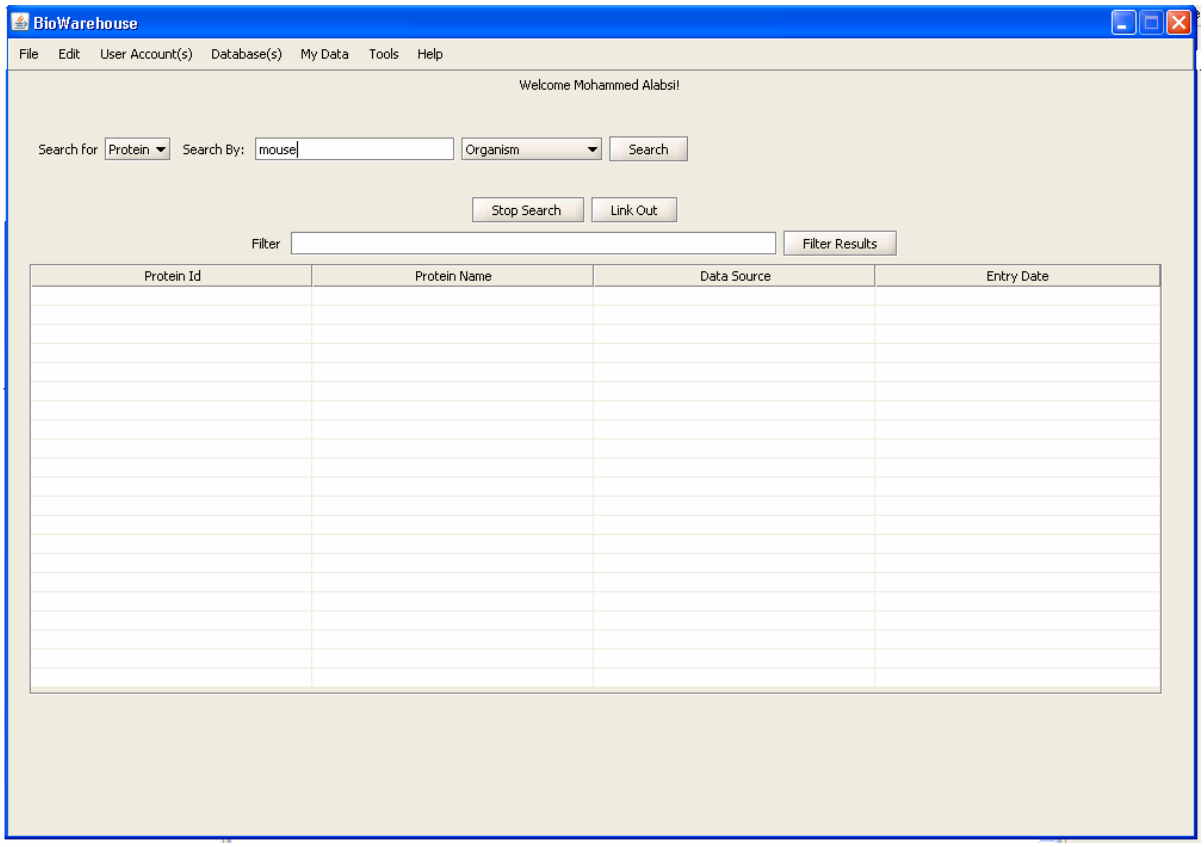

Figure 5.2: Java-Based Graphical User Interface.

5.2 Web Services

As mentioned above, web services work as the connector of the client-side GUI to the server-side. For now, there is one web service that is dedicated to handle all clients' requests. It contains functions to handle user login, receive query requests, search and retrieve user shared data, run a server-side tool, and many more. The web service is developed in Java using Apache Axis 1.4 library; which is an open-source web service framework that consists

of both Java and C++ SOAP server implementation, allowing the developer to generate and deploy SOAP servers and clients [18]. After little processing, the web services pass the client requests to the mediator, which figures out how to best address these requests. Other web services clients are implemented to search and retrieve data from public integrated data sources. One connects to PDB's web service, which provides structural data about proteins along with related publications; another connect to EBI's Web Service, which provides data from Interpro, UniprotKB, EMBL, Medline and other well known data sources (currently only Interpro and UniprotKB data is retrieved and parsed into the data warehouse) [15]. Although these services are useful to the client, the client does not directly interact with them. All web services run under an Apache Tomcat 5.0 server.

5.3 Data Sources

Several public, well known, data sources have been integrated into the system, which provide the users with a convenient way to access a large pool of interesting and trusted data. Because of time constraints, efforts were focused on integrating protein data sources only, with a vision to expand that in the future. After investigating public protein data sources, it was found that Protein Data Bank (PDB), Universal Protein Resource (UniProt), and InterPro were the most commonly known, trusted protein data sources. Hence, they were the data sources selected in this effort. The following is a short description of each data source:

- *Protein Data Bank (PDB)* is a worldwide depository of information about three-dimensional structures of large biological molecules, particularly proteins and nucleic acids. A variety of information is associated with each structure including sequence details, atomic coordinates, crystallization conditions, 3-D structure neighbors computed using various methods, derived geometric data, structure factors, and 3-D images [5].
- *Universal Protein Resource (UniProt)* is a central repository of protein sequences and functions created by joining the information contained in Swiss-Prot, TrEMBL, and PIR. The UniProt Consortium is comprised of the European Bioinformatics Institute, the Swiss Institute of Bioinformatics, and the Protein Information Resource [2].

- *InterPro* is an integrated documentation resource for protein families, domains and sites. InterPro combines a number of databases that use different methodologies and a varying degree of biological information on well-characterized proteins to derive protein signatures. Member databases include: PROSITE, Gene3D, PANTHER, PIRSF, Pfam, SMART, SUPERFAMILY, TIGRFAMs, and PRINTS [1].

Information retrieved from these data sources is returned as text, formatted in the standard format used by the data source. Once received, the mediator parses the text and stores it into the data warehouse as objects, which is usually returned to the user. Since each data source has its own data format, it was necessary to build a parser for each data source.

5.4 Gene Ontology (GO)

Gene Ontology is a structured, controlled vocabularies and classifications that cover several domains of molecular and cellular biology; freely available for community use in the annotation of genes, gene products and sequences. GO is divided into three sub-vocabularies: one describes the *molecular function* of gene products, one describes their roles in *biological processes*, and the third specifies their localization to *cellular components***.** Each GO term consists of a unique alphanumerical identifier, a common name, a definition, and possibly synonyms. Relations like: “is_a” and “part_of” are defined between GO terms, and are used to find relations between biological entities mapped to these terms (Figure 5.3) [3].

GO is used to search for related entities and find results that are semantically similar, and hence improve the search result. Currently, we are only using “is_a” relations. GO data is obtained by downloading GO file from gene ontology’s FTP server. A parser reads the files content and stores it in the data warehouse. Figure 5.3 shows an example GO entry for id 0002273, taken from a GO file. It shows the name of the biological entity, its category (namespace), a brief description about the entity, and a list of the gene ontology ids that are synonyms of this entity.

```
[Term]
id: GO:0002273
name: plasmacytoid dendritic cell differentiation
namespace: biological_process
def: "The process whereby a relatively unspecialized hemopoietic precursor cell acquires
the specialized features of a plasmacytoid dendritic cell." [GOC:add, PMID:15990333,
PMID:16174108]
is_a: GO:0002270 ! plasmacytoid dendritic cell activation
is_a: GO:0002521 ! leukocyte differentiation
```

Figure 5.3: Example Gene Ontology entry.

5.5 Data Warehouse

The data warehouse makes use of two components, namely MySQL and Hibernate, to store data on the server.

- *MySQL*: is a multithreaded, multi-user, relational SQL Database Management System (DBMS). Version 5.0.27 is used.
- *Hibernate:* An object-relational mapping solution for Java that maps an object-oriented domain model to a traditional relational database. Chapter 4 provides a detailed description about Hibernate basics and how it works. Version 3.2 of Hibernate is used.

The data warehouse is used to cache data retrieved from the data sources to improve performance. We have currently set the amount of time data is cached for to 7 days; which can be changed easily once we feel the need to in the future. The data warehouse is also used to store Gene Ontology (GO) data, which is queried to find semantically related results. Finally, it stores information about user accounts, and the databases users share with others; particularly, it keeps track of the type of data stored in each shared table (DNA, RNA or Protein), and the data attribute that each column in the table maps to (i.e. Protein Name, Pathway Name, etc…). This way, the system keeps track of which columns in which tables need to be queried to return the results requested by the user.

5.6 Server Tools

The system does not only focus on providing users with integrated biology data, but also with interesting tools that can be used to analyze the data retrieved. To do so, tools are placed on the server, and are made available to be used by all users. The process of adding tools to the server has been simplified so that adding any future tools would take very little time and effort. Specifically, there are three steps to add tools to the server:

- The tool should implement a simple interface, which contains five methods: three methods provide information and description about the tool and its usage, one method specifies the arguments needed by the tool, and one method used to invoke the tool (after the arguments being set).
- The tool is then placed into a JAR, which in turn is placed into a “tools” folder under Tomcat’s home directory.
- A method in the server code is called, which checks the existing tools, from the “tools” folder, and reloads these tools to the server’s memory. This results in that all new connections by clients will see the new list of tools.

Besides the speed and easiness of use, this mechanism allows the addition of tools that are not developed in Java, by simply building a Java wrapper on top of it. Therefore, tools developed in C#, for example, can be easily incorporated into the system as well. Currently, there are two tools that reside on the server: the first is a simple global sequence alignment tool that is based on Needleman-Wench's algorithm; and the second tool, called “ZiFit”, assists in the design of zinc finger proteins by identifying sites in DNA sequences for zinc finger protein design. More information about these tools and how to use them is described in chapter 7.

5.7 Mediator

The mediator is the component responsible for determining how to best handle user requests, for querying data sources and parsing their results, and for communicating with user shared

database and server tools. Hence, the mediator is considered the glue that brings the integration process together.

As a user query is passed to the server, it is immediately handed over to the mediator, which determines whether the data sources need to be queried or whether querying the server's data warehouse is sufficient enough to return the required results. To do so, the mediator checks whether the query has been submitted within the last week (7 days have been chosen because it would provide the system with enough cache power, and at the same time the system can ensure relatively up-to-date data). If the results are cached, the data warehouse is queried and the results are returned to the client. Otherwise, the mediator determines which data sources need to be queried, and in turn queries are sent to the data sources of interest. Upon receiving the results from the data sources, the results are parsed and cached into the data warehouse as objects, and returned to the client. In both cases, the mediator also queries user shared data for any possible matches. For every result, the mediator queries the Gene Ontology to find synonym results and return those as well, hence ensuring more complete and rich results. The mediator also acts as a middle-man when receiving user requests to run server tools: it runs the tool requested by the user, passing the needed arguments to it, and once the tool finishes execution it get the result from tool and passes it to the client.

Next chapter will show how the warehouse system is implemented, and explain how the system is used.

6. SYSTEM USAGE

This chapter describes the features of the system, and how it can be used by interested biologists. Mainly, the focus will be on: creating and managing user account, searching and exporting data, sharing data, and finally using client and server tools.

6.1 User Accounts

Most of the features of the system can be used by people who are not registered nor have a user account; so a guest user can search for data from public sources, export data to local files, and run client and server tools. Still, in order to obtain access to user shared data, a user must have an account. In the current implementation, singing up is free, and it can be done by simply filling in a sign up form as shown in Figure 6.1. From the “User Account(s)” menu, choose “create new account”. After choosing a user name, filling in your name, email and password, and clicking submit, the user account will be ready and the user can sign in right away.

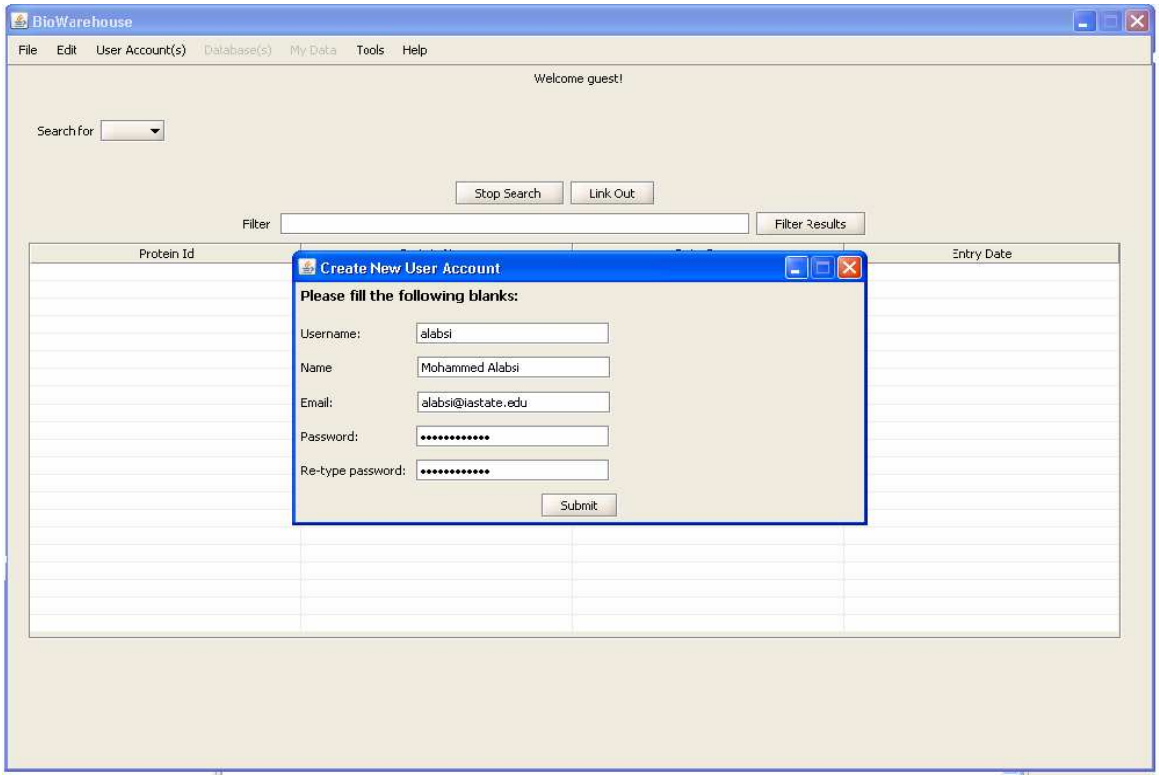

Figure 6.1: Creating a user account.

To login, the user clicks on “Login” under the “User Account(s)” menu; a login window will appear. Then, the user enters his/her username and password, and clicks submit. If the username and password provided match a record in the system database, the login will be successful and the user’s name will be shown on the top of the GUI (Figure 6.2).

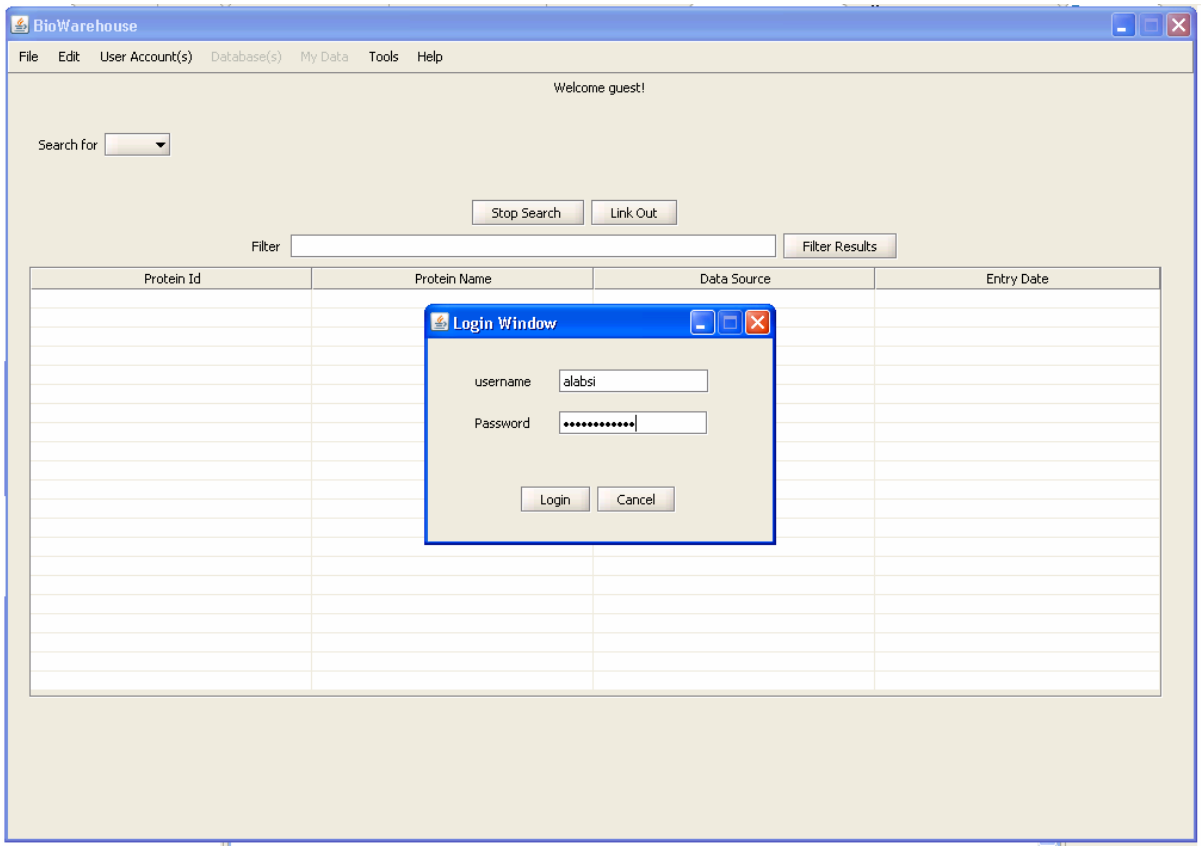

Figure 6.2: User logging into the system.

Once logged in, the user can try all of the features of the system, which are shown in the other section of this chapter.

6.2 Searching for data

To search for data, the user first specifies whether he/she is looking for Protein data, DNA data or RNA data. Then, the search term is entered and the search attribute is selected. For example, as in Figure 6.3, the user may choose to search for proteins whose species is “mouse”. In this case, mouse is the search term and search attribute is “species”. If results are found, the GUI will show a counter, below the results table, to describe how many results have been found so far. In case the user wants to stop the search before all possible results are explored, he/she can press the “Stop Search” button.

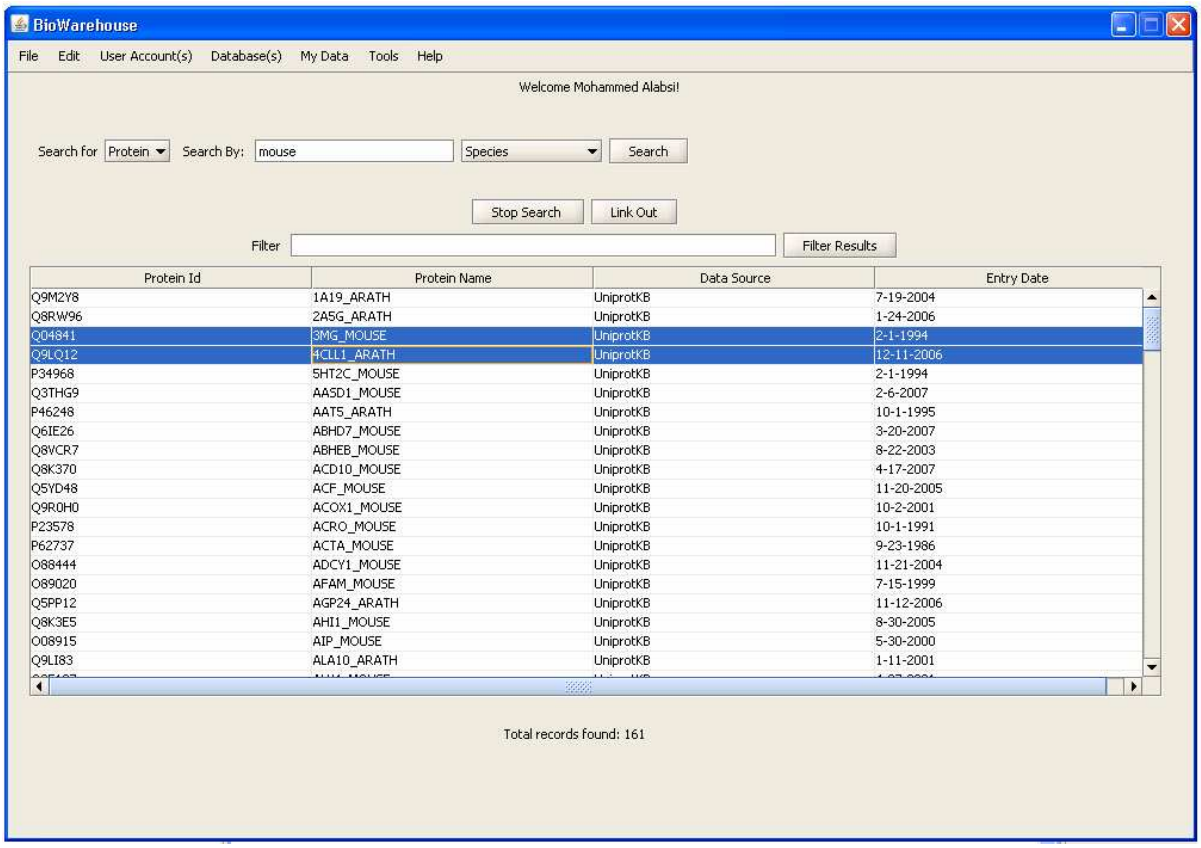

Figure 6.3: Search for protein data.

After receiving the results, the user can customize the columns shown; for example “Sequence” and “Keywords” can be added to the results table as in Figure 6.4. Also, rows can be ordered by a column of the user’s choice, or filtered by a keyword entered by the user. If the entry is from Interpro, UniprotKB or PDB, the user can open the web browser to a page from the data source that provides data about the entry selected. Finally, chosen entries can be exported to Excel files, or the default file format of the data source where the entry came from (if the entry is not from data shared by other users).

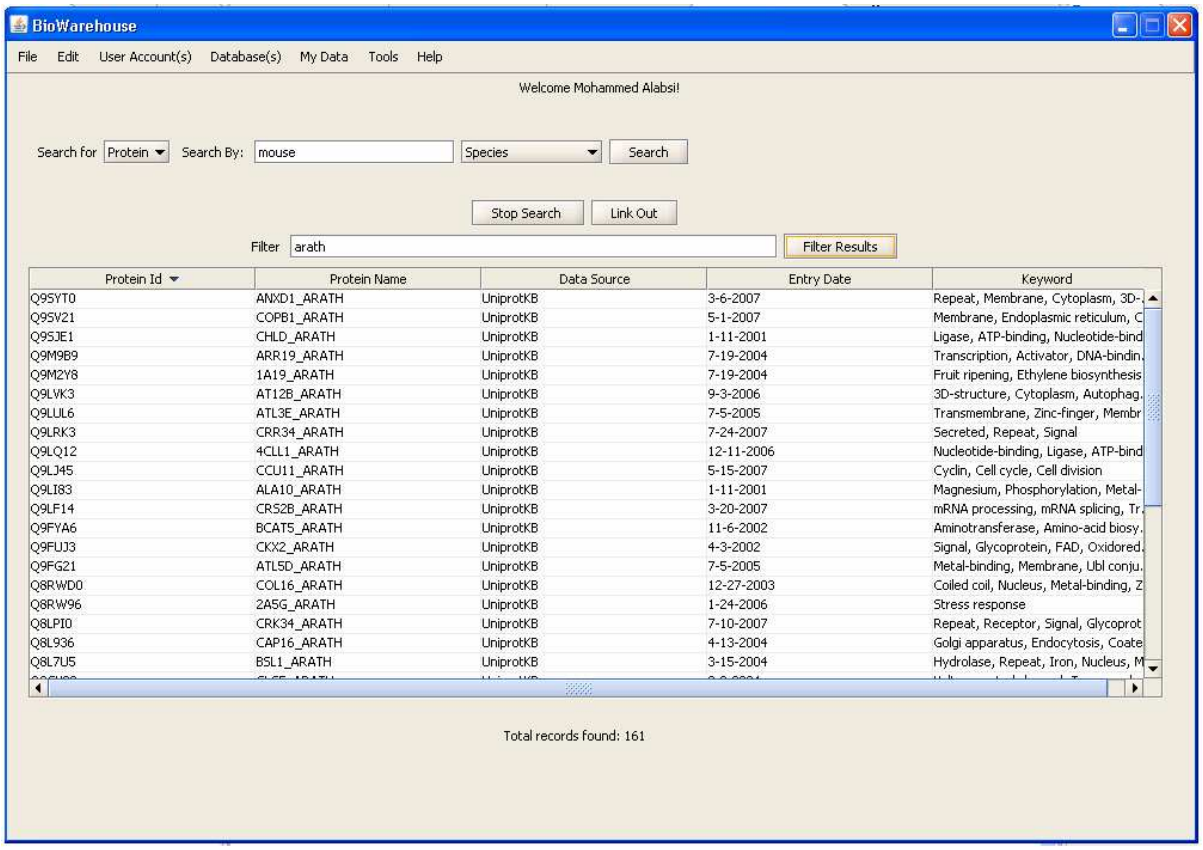

Figure 6.4: Result Set ordered by Protein Id, and filtered using the keyword “arath”. Also, the column “Keywords” is added to the results table.

6.3 Sharing User Data

Users can also share data with each other. To do so, the user first creates a “database profile”, which is simply the information needed in order for the system to be able to access the database (or part of it) to be shared; information like database name, database URL, username and password, and DBMS type (Figure 6.5). Once the profile is created, the user can choose table(s) from the database to share with others, as shown in Figure 6.6.

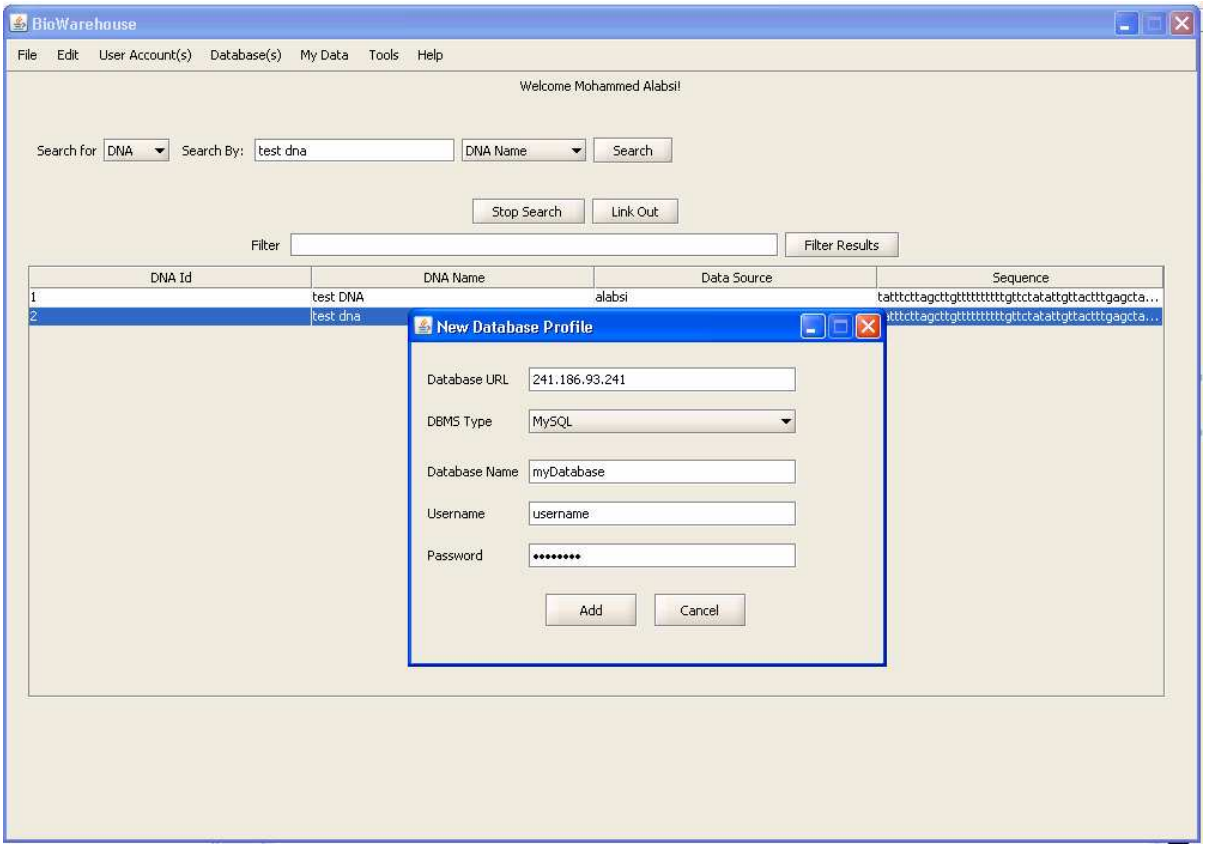

Figure 6.5: Creating database profile.

In order to do so, the user needs to provide information about the domain of data stored in the table (Protein, DNA, or RNA), and the type of data stored in each column, as shown in Figures 6.7 and 6.8. For example, the table may contain protein data, and its columns contain data about Protein Name, Protein Id and Gene Name. Once the table information is provided, the table is available for other users to query and retrieve data from.

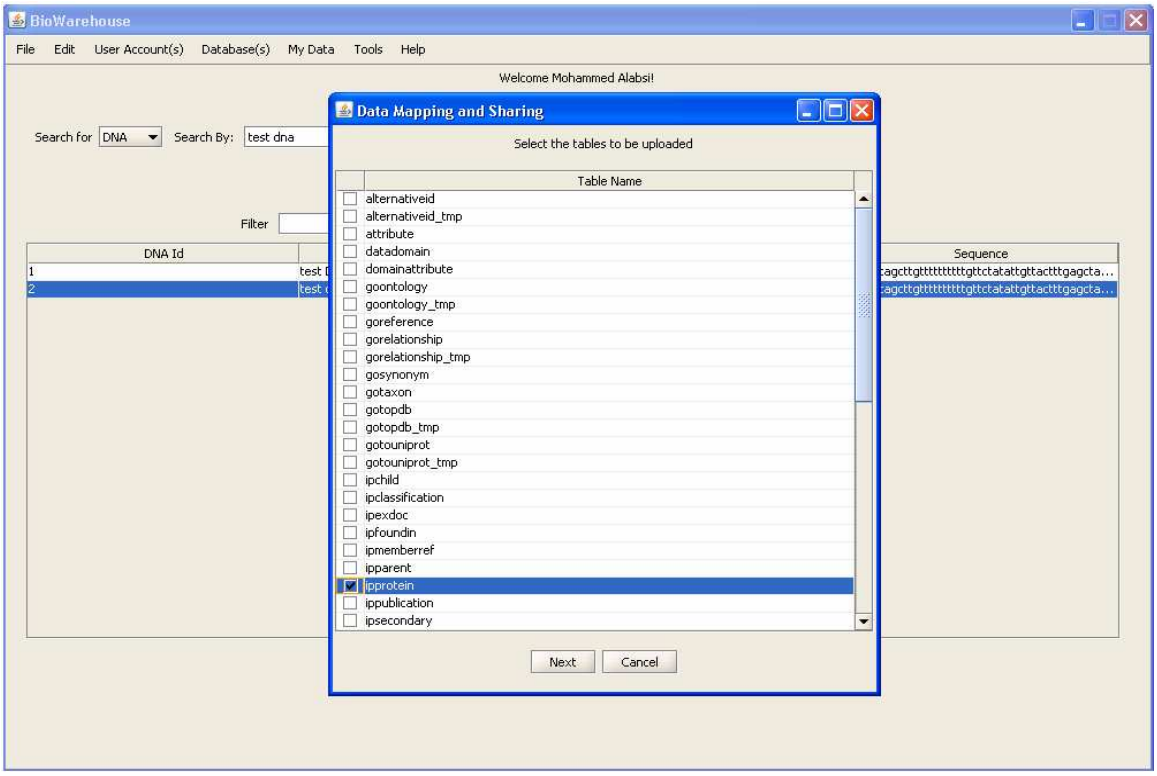

Figure 6.6: Choosing table from the database to share with other users.

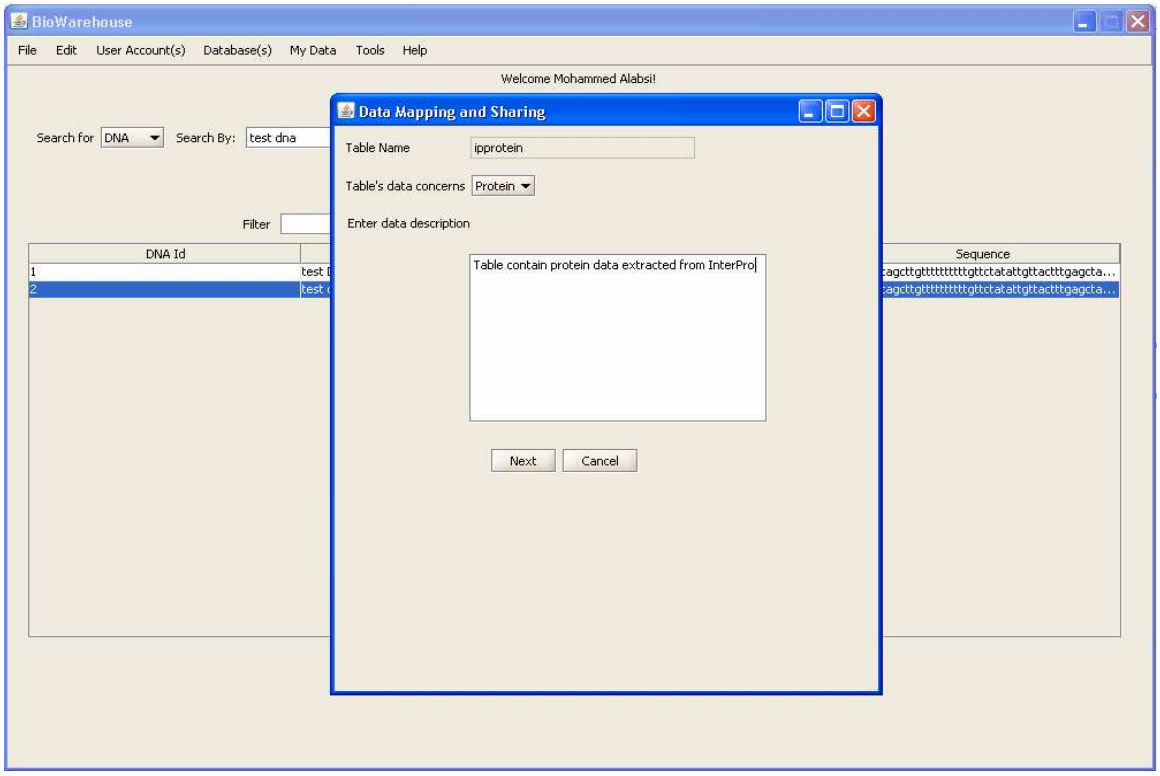

Figure 6.7: Providing description about the table to be shared and its data.

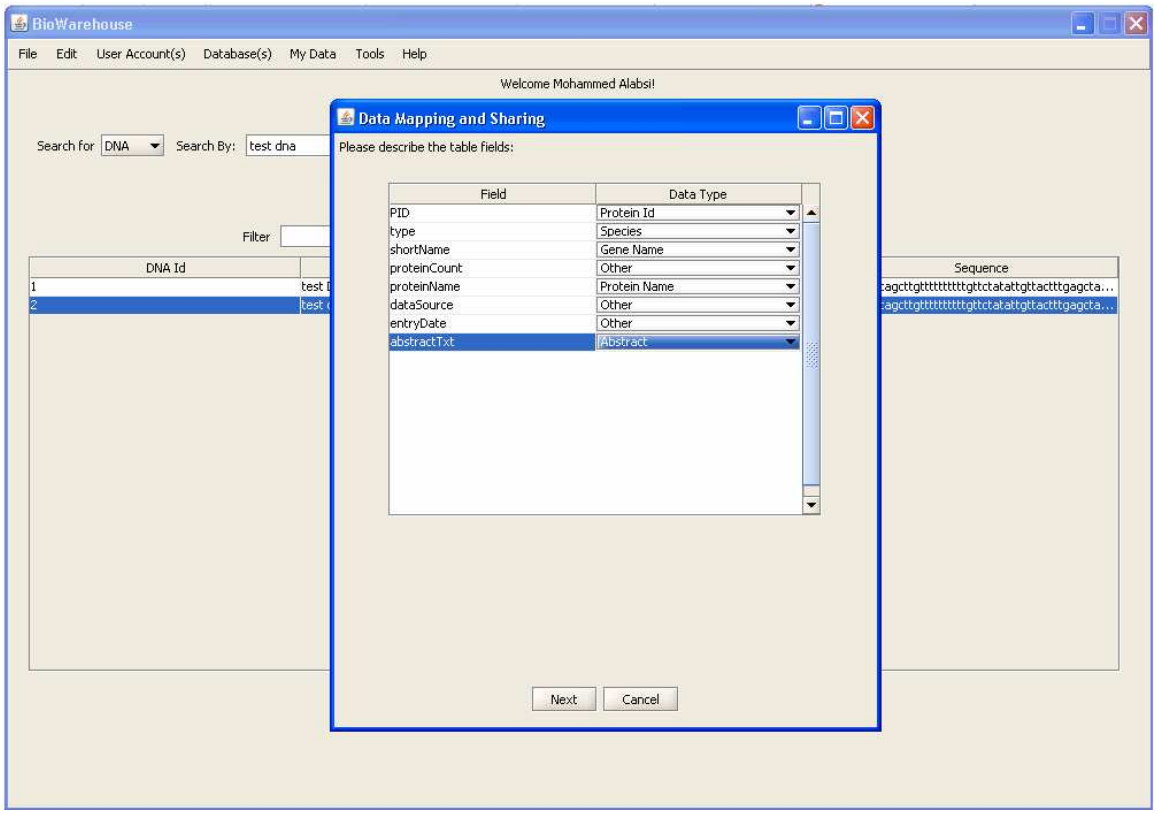

Figure 6.8: Mapping each column to the data type of the values it stores.

Anytime after its creation, database profiles can be edited or deleted. Similarly, shared table's mapping information can be changed, and the user can specify the table as not to be shared with others later, if desired. Finally, a report of the currently shared data can be viewed by choosing "view all tables properties" under the "My Data" menu, as shown in Figure 6.9.

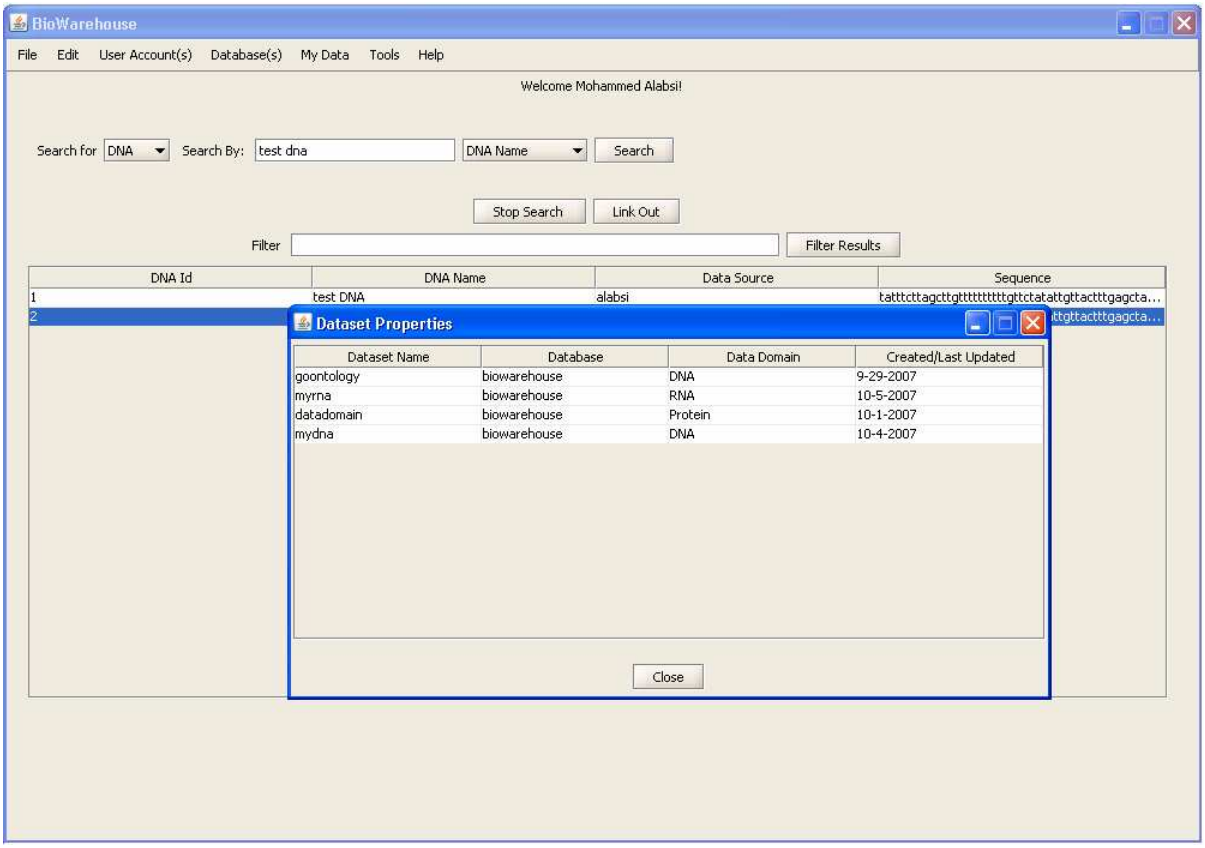

Figure 6.9: Viewing tables shared by the user.

6.4 Tools

There are two groups of tools that are available to the users: local, client based tools and server-side tools. Users can add their own tools to the first group, and plug in their own tools to the GUI, by wrapping each tool with a Java class that extends the “BioWarehousePlugin” class. Particularly, the wrapper class specifies the constructor of the tool, where an object of that tool is generated. It also defines a method that is called when the tool is being chosen by the user. Once the wrapper class is created, it is placed into a JAR which in turn is placed into the “plugins” folder. After restarting the client GUI, the user can see the tool added under the “Tools” menu, which can be run by selecting it from that menu.

The client GUI comes with a plug-in for an application called “PubMed Assistant”, which is a Java program that allows biologists to search for publications based on keywords provided by the user. By selecting an entry of interest from the results table, and choosing PubMed Assistant from the “Tools” menu, a window will appear that will ask the user to specify the

column that will be used to search for publications (Figure 6.10). Once that column is chosen, PubMed Assistant will appear, with the value of that column used to search for publications.

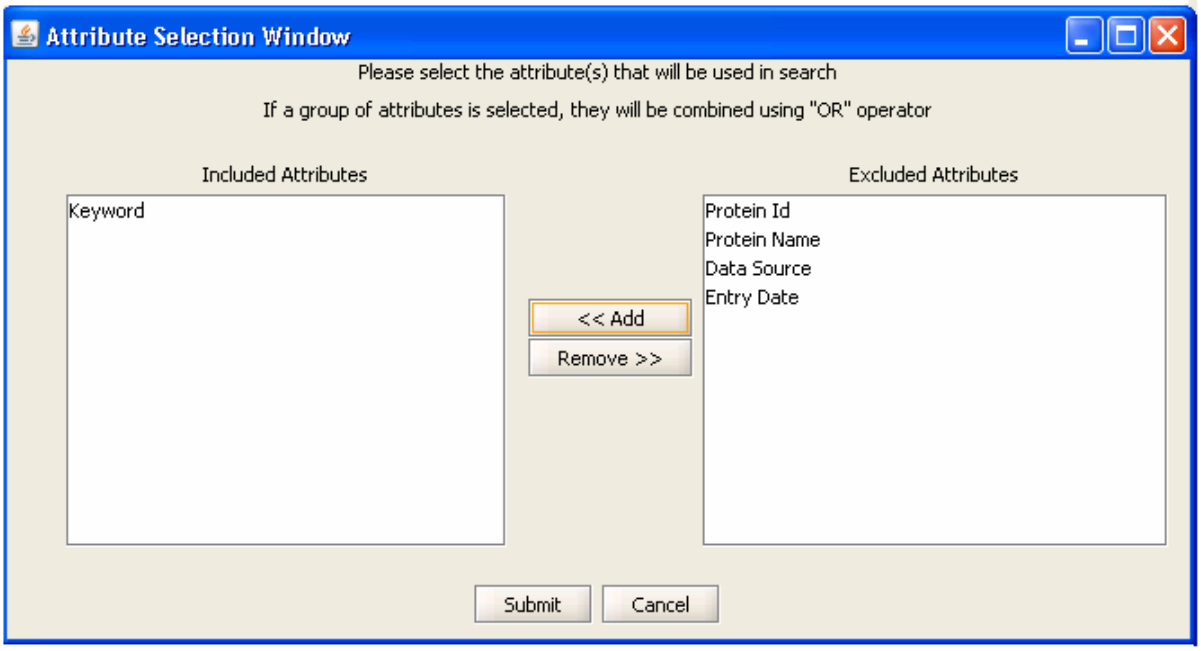

Figure 6.10: Choosing the column(s) whose value will be used to search for publications in PubMed Assistant.

The result of the search will be a list of publications, where the PubMed Id, publication title, and year shown (Figure 6.11). When a publication is selected, it's abstract and complete author list is shown, with the ability to link to the PubMed page that contains the chosen publication.

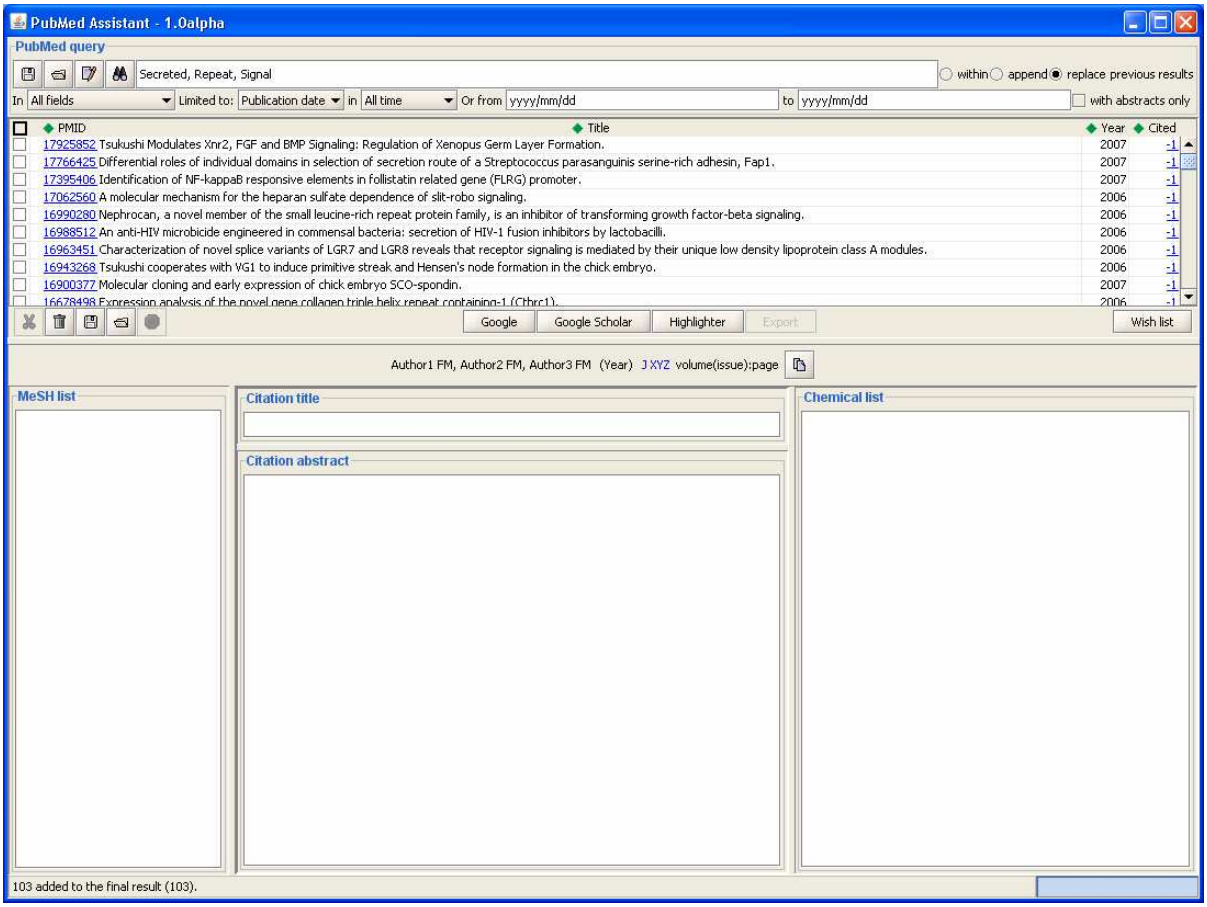

Figure 6.11: PubMed Assistant showing the results of search submitted by a user.

Another group of tools will reside on the server, and will be available for all clients to execute and receive the results through the GUI. Adding tools to the server is very straight forward, and is similar to the process of plugging in tools to the GUI. First, the tool is wrapped with a Java class that implements the “BioTool” interface; which contains five methods: three methods provide description and information about the tool, one method describes the required arguments for the tool, and the final method is used to run the tool. The wrapper class is then placed in a JAR, which is in turn put in a “tools” folder under tomcat’s home directory. Finally, a method on the server-side is called to reload the existing tools, which now include the new tool.

A server tool that currently incorporated into the system is ZiFit, which helps designing zinc finger proteins by identifying sites in DNA sequences for zinc finger protein design. Much like any other server tool, the tool has its sub-menu under the “Tools” menu. The tool’s

submenu is composed of three items: one runs the tool when clicked on; the second is used to pass and set arguments needed by the tool; and the third item shows a brief description of the tool and how it works.

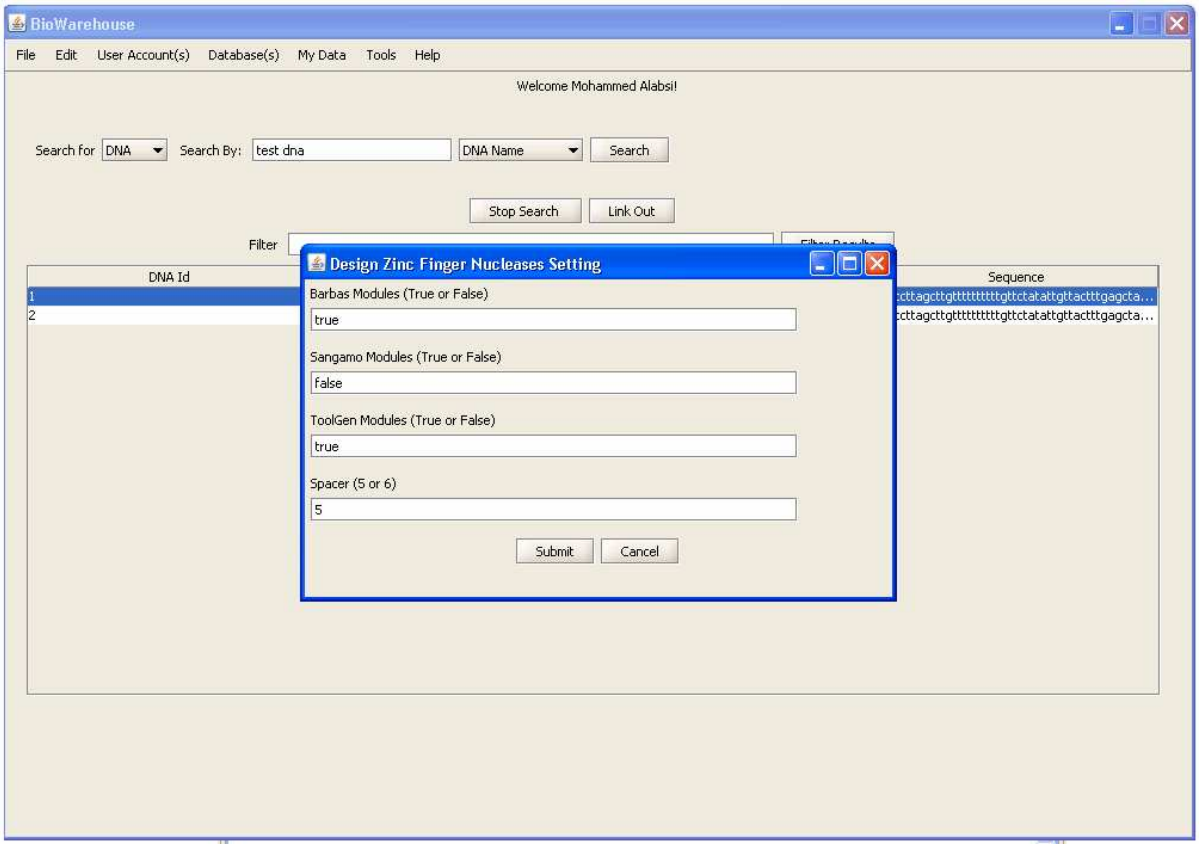

Figure 6.12: Setting up the required arguments for the server tool ZiFit.

Figure 6.12 shows the arguments setting window for ZiFit; it shows that four arguments are required, while showing the type of value needed for each argument. The first three arguments specify whether the Carlos Barbas module, and the Sangamo BioSciences Inc. modules, and ToolGen Inc. modules will be used in identifying target sites. The last argument specifies the number of spaces to be considered when looking for sites. Once, the arguments are provided, with valid values, the user can run the tool on the chosen DNA sequence. Figure 6.13 shows the result of running the ZiFit tool, which can be exported to a local file.

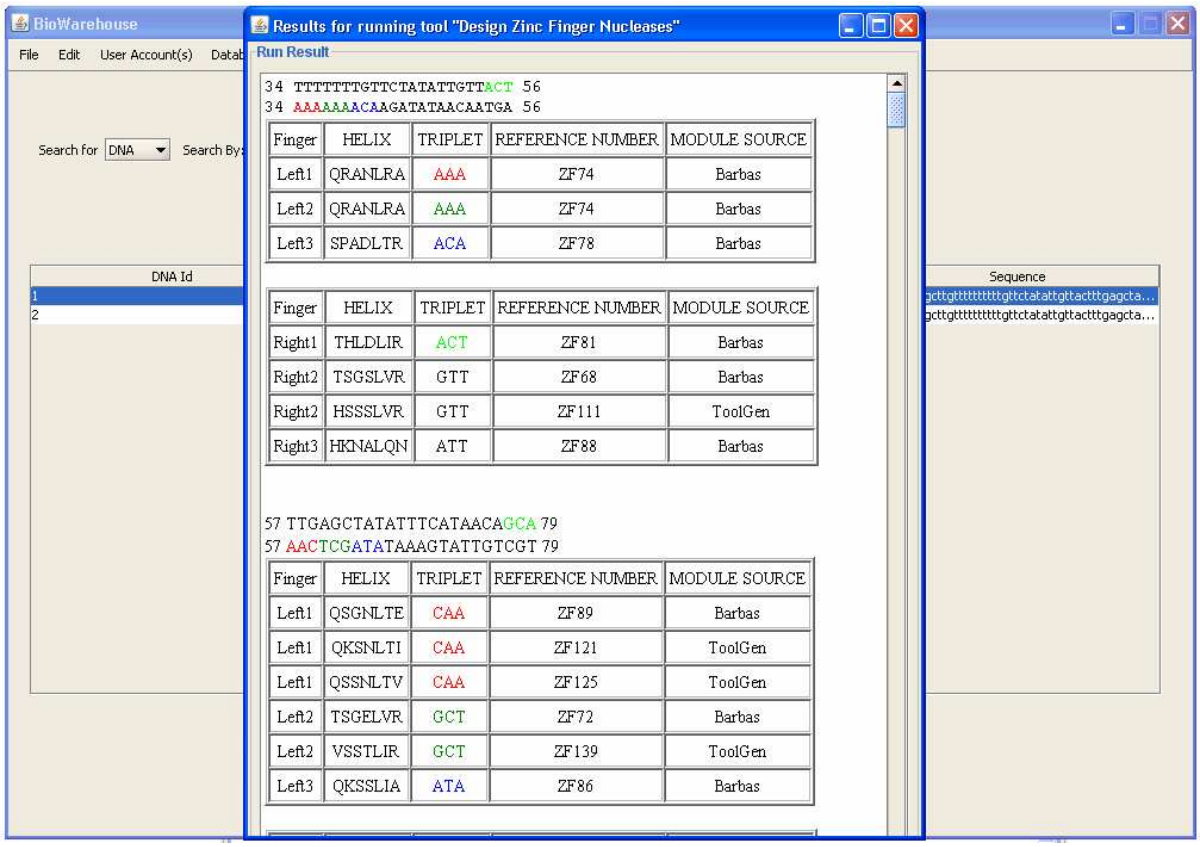

Figure 6.13: The result of executing ZiFit on the chosen DNA sequence.

7. CONCLUSION AND FUTURE WORK

Biological data integration is problem that the biology community continues to face; and as the size and number of available databases increases, the need for efficient and powerful solutions will increase along with it. In this project, we have introduced a solution that aims not only at handling the traditional issue of just gathering data from distributed sources, but rather ensuring that semantically similar data (discovered using Gene Ontology) are provided as well. Novel to such a system, efforts have been focused on integrating and sharing biological tools as well as allowing users to share data between each other. The system design was based on the client-server paradigm, where the main features of the systems resided on the server-side, and the client is used as an interface to the system and as a mean to pass user requests and queries. This provides the flexibility to change features and functionalities in the system, by applying changes to the server, without the need to update the client GUI. Client and server communicate with each other using SOAP-based web services. The system design and goals describes in chapter 5 have been met, and the system have been implemented successfully.

In the future, some features or functionality could be added to the system to enhance its flexibility and versatility:

- More public data sources could be integrated, particularly ones that provide RNA, DNA and publications data.
- Users would have the ability to browse Gene Ontology, and search data using GO terms. Also, users would have their own customizable ontology, where they can add new terms or modify existing ontology.
- Users would have a “friends list”, and they could specify the level of privacy for each dataset shared. For example, a dataset can be “public”, where it is available to all other users; or “protected”, where it is only available to friend users.
- Building an AJAX-based web interface to the warehouse system.

REFERENCES

[1] Apweiler R, et al, "The InterPro database, an integrated documentation resource for protein families, domains and functional sites", Nucleic Acids Research, Vol. 29, 2001.

[2] Apweiler R., et al, "UniProt: the Universal Protein knowledgebase", Nucleic Acids Research, Vol. 32, 2004.

[3] Ashburner M., et al, "Gene Ontology: tool for the unification of biology", Nature Genetics, Vol. 25, 2000.

[4] Baker P., et al, "TAMBIS - Transparent Access to Multiple Bioinformatics Information Sources", in Proc. 6th International Conference on Intelligent Systems for Molecular Biology, 1998, pp. 25-34.

[5] Berman H.M., et al, "The Protein Data Bank", *Nucleic Acids Research*, Vol. 28, 2000.

[6] Davidson S., et al, "K2/Kleisli and GUS: Experiments in Integrated Access to Genomic Data Sources", IBM Systems Journal, Vol. 40, No. 2, 2001.

[7] Davidson S., Buneman P., Harker S., Overton C., Tannen V., "Transforming and Integrating Biomedical Data Using Kleisli: A Perspective," SIGBIO Newsletter, Vol. 19, pp.8-13, Aug.1999.

[8] Galperin M., "The Molecular Biology Database Collection: 2006 update", *Nucleic Acids Research*, Vol. 34, 2006.

[9] Galperin M., "The Molecular Biology Database Collection: 2007 update", *Nucleic Acids Research*, Vol. 35, 2007.

[10] Haas L, et al, "DiscoveryLink: A System for Integrated Access to Life Sciences Data Sources", IBM Systems Journal, Vol. 40, No. 2, 2001.

[11] Hernandez T., Kambhampati S., "Integration of Biological Sources: Current Systems and Challenges Ahead", SIGMOD Record, Vol. 33, No. 3, Sept 2004.

[12] “HIBERNATE - Relational Persistence for Idiomatic Java”, 2006, http://www.hibernate.org/hib_docs/v3/reference/en/html/

[13] Lafon Y., “Web Services @ W3C”, Sept 2007, http://www.w3.org/2002/ws/.

[14] Lee T., et al, “BioWarehouse: a Bioinformatics Database Warehouse Toolkit”, BMC Bioinformatics, Vol. 7, pp. 170-184, March 2006.

[15] Pillai S., et al., “SOAP-based services provided by the European Bioinformatics Institute”. *Nucleic Acids Research*, Vol. 33, April 2005.

[16] “Universal Description Discovery and Integration”, Nov 2007, http://en.wikipedia.org/wiki/Universal_Description_Discovery_and_Integration.

[17] “W3C SOAP Activities”, 2007, http://www.w3schools.com/w3c/w3c_soap.asp.

[18] “Web Services – Axis”, 2006, http://ws.apache.org/axis/.

ACKNOWLEDGEMENTS

This work would not have been possible without the support of many people.

First, I would like to thank my major professor Dr. Les Miller for his guidance and support while working on this project, and for being a source of motivation and inspiration throughout my graduate education.

I would also like to thank Dr. Tavanapong and Dr. Dobbs for their willingness to serve on my POS committee and for their help when needed. Thanks also to Linda Dutton for her assistance and guidance in many matters.

Last but certainly not least, my deepest thanks go to my parents, Samir and Randa, for their continuous love and support that helped me become the person I am.

Mohammed Alabsi