J2SE, JSP & JavaBeans
Web Demonstration of SWKB Program

Bachelor report presented to
CUI
Geneva University

by

Arnaud Jotterand

Supervisor:
Prof. Stephane Marchand-Maillet

Geneva, October 2005

Abstract

This project is a web application that allows to demonstrate a research group Viper's software, the Semantic Web Knowledge Base (SWKB).
The idea of this software is that a collection is annotated with a description structured using a DEVA model. The SWKB, as a forward chaining reasoning engine, then allows rich queries, thanks to the relationships created between the DEVA-based annotation and an OWL-compliant ontology.

This project is based on the Java technology, Java 2 Platform, Standard Edition (J2SE). The application uses JSP pages and JavaBeans, and can be deployed on a Tomcat Server.

Contents

List of Figures

Chapter 1

Introduction

Many research groups work around the world, specially in the domain of computer sciences. It is very important for the scientific community that all these groups publish their results. The best way actually to distribute these informations is to publish them on the internet; so is a new information quickly known from scientifics around the world.

The research group Viper has developed a new software, the Semantic Web Knowledge Base (SWKB). It is possible to explain the functionalities of this software with text and screenshots on the website of the research group. However, the best way to expose the functionalities of this software is to create an online demonstration of this software, that allows the user to use the developed software on the internet, directly on the website. This online demonstration is the goal of this project.

Chapter 2

Description of the project

In this chapter are described the main lines of this project, his purpose, and the reasons of doing it.

2.1 The SWKB Program

2.1.1 Principle

The research group Viper has developed a program called Semantic Web Knowledge Base (SWKB). The idea is that a collection is annotated with a description structured using the DEVA model, defined by the Viper group. This DEVA model is an annotation container; it is an RDF-compatible extension of the Dublin Core (DC). It extends the "element" field of the DC into a flexible RDF-based structure. The SWKB, as a forward chaining reasoning engine, then allows rich queries, thanks to the relationships created between the DEVA-based annotation and an OWL-compliant ontology.

2.1.2 Software

The SWKB program allows to search for images in a collection, using rich queries. A software has been developed by the Viper group to show the features of SWKB; this software, written in Java, allows the user to submit a query (using the DEVA syntax), and to see the images (and their annotations) resulting from this request.

This software is powerful, but presents two disadvantages:

Uneasily accessible This software is written in Java, so it is easily portable. However, when a user wants to try this software, he has to download it first; it would be better to directly show the program features without downloading.

Unfriendly user query select The software proposes a default query (with DEVA syntax); it is possible to modify this query in order to see different results, but the problem is that the user does not know the DEVA syntax, which is not very instinctive. Without a knowledge of this syntax, the user can only try the default query, and can not really notice how powerful the SWKB is.

2.2 Internet Version

The best way to expose the program's features is then to develop an internet version, which can be accessible directly from the Viper's website. This internet version is the subject of this project.
In order to solve the second disadvantage, the internet version should propose an attractive solution to the problem of the way to select the query; it would be easy for the user to try different queries and to observe their results, without any knowledge of the DEVA syntax. The solution is

- to propose different queries expressed in litteral english
- to automatically give the translation in DEVA syntax
- to allow the user to edit then this query expressed in DEVA syntax

Obviously, the internet version has to propose the same functionnalities as the possibility of posting the images in real size or the possibility of seeing the corresponding annotations.
Thus the main webpage of this project should comport three parts:

1. A part concerning the choice and the editing of the request
2. A part showing the results (found images)
3. A part exposing the details (like the annotations) of a selected image

Chapter 3

Implementation

In this chapter are given a brief overview of the used technologies, a description of the project's implementation as well as some tips to update this application.

3.1 J2SE Technology

This project uses Java 2 Platform technology, in particular the Java Server Pages and the JavaBeans. Before explaining the project's implementation, a brief theorical overview of these two Java technologies is given.
This part is just an overview of these technologies; for more detailed information, see [1] and [2], from which this section is strongly inspired.

3.1.1 Java Server Pages

JavaServer Pages (JSP) technology allows the user to easily create web content that has both static and dynamic components. JSP technology makes available all the dynamic capabilities of Java Servlet technology but provides a more natural approach to creating static content.

A JSP page is a text document that contains two types of text:

static data The static content is simply written as if it were a page that consisted only of that content; it can be expressed in any text-based format such as HTML, SVG, WML, and XML, but the default format is HTML. It is possible to use another format, just by adding a page

directive with the contentType attribute set to the content type; the purpose of the contentType directive is to allow the browser to correctly interpret the resulting content.

dynamic content The dynamic content is created by accessing Java programming language object properties; it is possible to access a variety of objects, including JavaBeans components. JSP technology automatically makes some objects available (implicit objects), and it is also possible to create and access application-specific objects. The implicit objects are created by the web container and contain information related to a particular request, page, session, or application. The application-specific objects allow to encapsulate application behavior in objects; the main way to use them within a JSP page is to use JavaBeans components, that can be set or accessed by JSP standard tags or expression language.

A JSP page services requests as a servlet; thus, the life cycle and many of the capabilities of JSP pages (in particular the dynamic aspects) are determined by Java Servlet technology. When a request is mapped to a JSP page, the web container first checks whether the JSP page's servlet is older than the JSP page. If the servlet is older, the web container translates the JSP page into a servlet class and compiles the class. During the translation phase each type of data in a JSP page is treated differently: static data are transformed into a code that will emit the data into the response stream, and JSP elements are treated separately (for example directives are used to control how the web container translates and executes the JSP page, custom tags are converted into calls to the tag handler that implements the custom tag, ...).
During development, one of the advantages of JSP pages over servlets is that the build process is performed automatically.

In order to use dynamic content within JSP pages, the preferred mechanism is the custom tags. They can be used to perform a wide variety of dynamic processing tasks, including accessing databases, using enterprise services such as email and directories, and implementing flow control. The JavaServer Pages Standard Tag Library (JSTL) encapsulates core functionality common to many JSP applications. Instead of mixing tags from numerous vendors in JSP applications, JSTL allows to employ a single, standard set

of tags. This standardization allows to deploy applications on any JSP container supporting JSTL and makes it more likely that the implementation of the tags is optimized. JSTL has tags such as iterators and conditionals for handling flow control, tags for manipulating XML documents, internationalization tags, tags for accessing databases using SQL, and commonly used functions.

3.1.2 JavaBeans

The JavaBeans API makes it possible to write component software in the Java programming language. Components are self-contained, reusable software units that can be visually composed into composite components, applets, applications, and servlets using visual application builder tools. JavaBean components are known as Beans. Components expose their features (for example, public methods and events) to builder tools for visual manipulation. A Bean's features are exposed because feature names adhere to specific design patterns. A "JavaBeans-enabled" builder tool can then examine the Bean's patterns, discern its features, and expose those features for visual manipulation.
Builder tools discover a Bean's features (that is, its properties, methods, and events) by a process known as introspection. Beans support introspection by adhering to specific rules, known as design patterns, when naming Bean features (the Introspector (in the API reference documentation) class examines Beans for these design patterns to discover Bean features), or by explicitly providing property, method, and event information with a related Bean Information class. Although Beans are designed to be understood by builder tools, all key APIs, including support for events, properties, and persistence, have been designed to be easily read and understood by human programmers as well.

The JavaBeans API provides a standard format for Java classes. Visual manipulation tools and other programs can automatically discover information about classes that follow this format and can then create and manipulate the classes without the user having to explicitly write any code. The main rules for writing Beans are:

- A bean class must have a zero-argument (empty) constructor
- A bean class should have no public instance variables (fields)

- Persistent values should be accessed through methods called getXxx and setXxx (excepted with boolean properties, that use a method called isXxx)

Although we can use JSP scriptlets or expressions to access arbitrary methods of a class, standard JSP actions for accessing beans can only make use of methods that use the getXxx/setXxx or isXxx/setXxx design pattern.

3.1.3 MVC Architectural Approach

There are many different architectural approaches for applications; one of them is the Model-View-Controler (MVC). The MVC architecture is a widely used architectural approach for interactive applications that distributes functionality among application objects so as to minimize the degree of coupling between the objects. To achieve this, it divides applications into three layers: model, view, and controller. Each layer handles specific tasks and has responsibilities toward the other layers:

Model The model represents business data, along with business logic or operations that govern access and modification of this business data. The model notifies views when it changes and lets the view query the model about its state. It also lets the controller access application functionality encapsulated by the model.

View The view renders the contents of a model. It gets data from the model and specifies how that data should be presented. It updates data presentation when the model changes. A view also forwards user input to a controller.

Controller The controller defines application behavior. It dispatches user requests and selects views for presentation. It interprets user inputs and maps them into actions to be performed by the model. In a web application, user inputs are HTTP GET and POST requests. A controller selects the next view to display based on the user interactions and the outcome of the model operations.

3.2 WebPage

3.2.1 Structure

The Model-View-Controller (MVC) architectural approach is a good approach to web applications; it allows web designers (who often are not programmers) to concentrate their work on the View layer, without having knowledge of how are implemented the other layers (implemented by Java programmers for example). The structure of this project is inspired from this architectural approach, but with some modifications. So is this application composed of two layers: the Model layer, and a layer which joins both View and Controller layers together. This choice has been made because the application was not so big, and the Controller layer was not an important part of it. However, it was useful (and almost necessary) to divide the application into two layers, because the SWKB program existed before this application, and the goal was to reuse the program just as it was to expose it's functionalities. The Model layer is then what handles with this program, while the View-Controller layer concerns the web application.

The contents of the application are the following:

JSP Pages A main page (*swkb.jsp*) handles with queries and their results, and can call a page with the details of an image (*detail.jsp*) or a help page used as a popup (*help.jsp*). An error page (*error.jsp*) is also defined in order to have a friendly way to indicates errors to the user.

JavaBeans A bean is used to handle with the SWKB program (*SwkbBean.java*), another to handle with queries (*QueryBean.java*), and the third one to handle with details of a given image (*DetailBean.java*).

WebDesign Files Some Javascript files, CSS file, image files are used to give a friendly design to the page.

Deployment Descriptor Element A *web.xml* file is used as a deployment descriptor for the application.

In order to build this application, the project contains also a *build.xml* file to use with Apache Ant[1].
The way of how these files are organised is shown in figure 3.1.

[1]For more details on this tool, see *http://ant.apache.org/*.

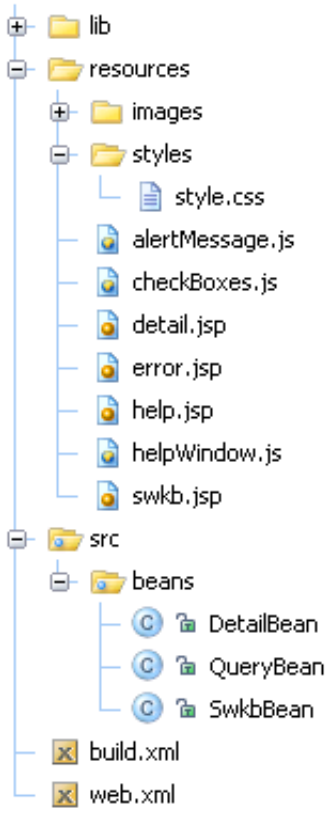

Figure 3.1. Tree Structure of the Project

3.2.2 View and Controller Layer

The View-Controller layer is implemented using the Java Server Pages technology. The static part of the JSP pages concerns the design of the webpage, while the dynamic part is used to update the presentation of the page and to handle with HTTP GET and POST requests. The pages use also some Javascript in order to handle with events such as *onkeypress* or *onchange*; these Javascript functions are used to put the query into the textarea without reloading the page.

The role of the Controller part is done with conditional tags of the JSP Standard Tag Library (JSTL). Using the JSTL in a JSP page requires to:

- put a page directive in the JSP page that indicates its use
- put the corresponding Tag Library Descriptor (TLD) and JAR files in the WEB-INF directory

The main page of this web application reloads itself with different request parameters. Depending on the request parameter, the Controller decides what actions the Model has to perform:

Empty request parameter The page is accessed without request parameter mainly when the user starts the session; the Model layer has then to dynamically find the different queries available in the file *queries.txt* (in order to be displayed by the View layer), and to proceed to the initialisation of the SWKB program[1].

Search request parameter The page is accessed with the *search* request parameter when the user clicks on the *Find Images*'s button; the Model layer has then to proceed to the research of the images corresponding to the request written in the textarea[1].

Image request parameter The page is accessed with the *image* request parameter when the user clicks on a resulting image in order to see its details; the Model layer has then to dynamically find the annotations corresponding to the corresponding image[1].

The Controller has then to give the user inputs to the Model; the way used in this application to do so is to work with JavaBeans. The Controller interprets the user inputs and then sets the corresponding properties of the JavaBeans using JSP tags. Then, when the Controller asks the Model to perform actions, the JavaBeans can use their own properties that contain the user inputs. The use of JavaBeans is also an easy way of storing data that can later be necessary.
The Controller decides also what to display in the page, depending on the request parameter:

- The alert message should be displayed only if some results are shown; this is the case when the request parameter is not empty.

[1]The way to use the SWKB program is explained later, in section 3.2.3, page 14.

- The section with the image's details should be displayed only if the user has chosen to see them; this is the case when the request parameter is *image*.

- The results section should be displayed only if the user has already done a search; this is the case when the request parameter is not empty.

One of the goal of this project was to reduce to a minimum the use of Java code in the JSP pages. Tag libraries reduce the necessity of embedding large amounts of Java code in JSP pages by moving the functionality of the tags into tag implementation classes. This is why, in order to render the content of the Model, the View part gets data from the Model using tags, such as JSP tags to get or set properties from JavaBeans, or JSTL tags to effectuate some loops over collections contained in the JavaBeans.

3.2.3 Model Layer

The Model layer is responsible for handling with the SWKB program, whose classes are given as .jar files in the *lib* directory (and also some useful files such as *.owl* files). The Java code necessary to use the SWKB program is totally contained in JavaBeans, in order to avoid Java code in the JSP pages.

The bean *SwkbBean.java* contains methods to initiate the program and to do the search for images. The initialisation consists mainly in loading the annotations contained in the *.owl* files into the program. In order to do the search for images corresponding to a given query, the bean has to call SWKB's methods with the query that has been stored in a bean's property by the Controller. The result of the search is a list of images' names; this list is stored in a property of the bean, so that the View can access it (using JSP tags).

The bean *QueryBean.java* is used to dynamically list the queries contained in the file *queries.txt*, so that the View can display them. When the Controller asks the Model to find these queries, the bean parses the queries' file and stores the list of queries in a property of the bean, so that the View

can access it (using JSP tags).

The bean *DetailBean.java* is used to dynamically find the annotations of a given image and to create a *.owl* file with them, so that the View can display it. When the Controller asks the Model to create this *.owl* file, the bean parses the file *galapagos_ann.owl* in order to find the annotation corresponding to the image selected by the user (the name of this image has been set in the bean property by the Controller). When the annotation is found, the bean creates a *.owl* file with it, and stores the name of this file in a property of the bean, so that the View can access it (using JSP tags).

The whole Model layer is thus contained in JavaBeans, entirely written in Java (which is easier for handling with a Java program). The JSP pages do not contain any Java code, so it is easier to update the design of the page without changing the Model; and reciprocally, it is easy to update the business logic without modifying the design of the page.
The coupling between logic and design has been reduced to the minimum.

3.3 Update Manual

The application has been developed in order to be relatively easy to update. It is possible to enrich it with new queries, or with other keywords to check. It is also possible to work with another images' database, for example. There are many possibilities of updating; the idea here is not to list them all, but just to give some tips for the main possible updates.

Add queries If we want to enrich the application with new queries, we just have to add them in the file *queries.txt* with the same syntax and the same punctuation (this is important because the parsing of the file is made by following the used rules); the bean will dynamicly add them to the list of queries.

Add keywords as checkboxes If we want to enrich the application with new keywords to check, we just have to add them in the main JSP page *swkb.jsp*, following the same rules as for the others; these rules are: to call the same Javascript function for the event and to give the keyword

as name of this checkbox. The Javascript function can then directly work with this new checkbox[1].

Add images in the database Now, the images treated by the SWKB program are located in the path *images/galapagos* of the application[2]. If we want to add images to this directory, there is nothing else to do, except verifying that these images are listed in the *.owl* files *galapagos.owl* and *galapagos_ann.owl*, so that SWKB can work with them; the web application will find these images without any problem. However, if we want to work with images that we put in another directory, we will have to modify the path used in the JSP pages to find the images: in the file *swkb.jsp* in both the details (line 159) and results (line 178) parts, and in the file *detail.jsp* (line 42).

Use another database It is possible to use an other database than the galapagos database with this application, but we have to be careful. The methods used to communicate with the SWKB program, in the file *SwkbBean.java*, indicate paths to the galapagos' database; they have to be modified for an other database. The parsings of files used in both *QueryBean.java* and *DetailBean.java* depend on how each file is written; in the case we use other files to use an other database, we have to modify these parsings in order to conform to the new files. However, the JSP pages do not have to change, because the database is related to the Model layer only.

The data files used by the SWKB program (and also with this web application), such as *queries.txt* or the *.owl* files, situated in the *lib* directory of the project, are put in the *WEB-INF/classes* directory of the application, so that the beans can access them[3]. If we need to add such files to the application, we should modify the *build.xml* file in order to put also these new files into the *WEB-INF/classes* directory (target *build*).

[1]The Javascript functions consider that there are 4 following inputs after the checkboxes (1 textarea and the 3 buttons), and one before the checkboxes (the select input). If we add some inputs before or after the checkboxes, we have to update the limits of the 'for' loops in the both Javascript functions situated in file *checkBoxes.js*.

[2]This path correspond to the path within the builded application; in the project (not builded), the directory *images* is situated in the directory *resources*.

[3]The beans use the code *getClass().getClassLoader().getResourceAsStream("nameOfTheFile")* to dynamically find the path of this directory on the server.

The *.owl* files created by the application to get the annotations of a given image (*annotationX.owl*) are stored in the main directory of the application in order to be displayed by the JSP pages. In order to avoid huge amount of such files on the server, the server should erase them periodicly.

These are just some possible updates. It is not possible to list here all the different updates, but the author remains available for any question regarding any update of this application.

Chapter 4

User Manual

In this chapter are given instructions on how to use the application.

4.1 Application's Main Page

4.1.1 Select Query

When we arrive on the webpage, we can see something similar to figure 4.1. We have then to select a query in order to see the features of the SWKB program. This can be done by two ways:

Selecting a predefined query There are some predefined queries listed in a select box (see figure 4.1). We can here select one of them by clicking on the chosen query; the corresponding DEVA query will appear in the textarea (see figure 4.1). The idea is here to ask SWKB for images corresponding to the selected query.

Selecting keywords There are some keywords displayed as checkboxes (see figure 4.1). When one or many of these keywords are checked, a DEVA query formed with these keywords appears in the textarea (see figure 4.1). The idea is here to ask SWKB for images whose annotation contains the checked keywords.

These two ways are exclusive; when we choose to check a keyword after selecting a query, for example, the DEVA query is replaced by the query formed by the keyword. In both ways, it is possible at any time to reset what has been done, simply by clicking the *Reset* button (see figure 4.1).

Once the query selected (and put into the textarea), it is possible to edit it manually before doing the search. However, we have here to follow the DEVA syntax; if not, the SWKB will not understand the query and the results will not be very interesting.

The figure 4.1 represents the basic situation of the webpage. Later, other parts can be displayed, but the part concerning the query will stay as explained above. The only difference is that a message can appear explaining that the query has been modified, and that shall not correspond to the parts displayed below any more. At any time, we can select another query just by proceeding as explained above.

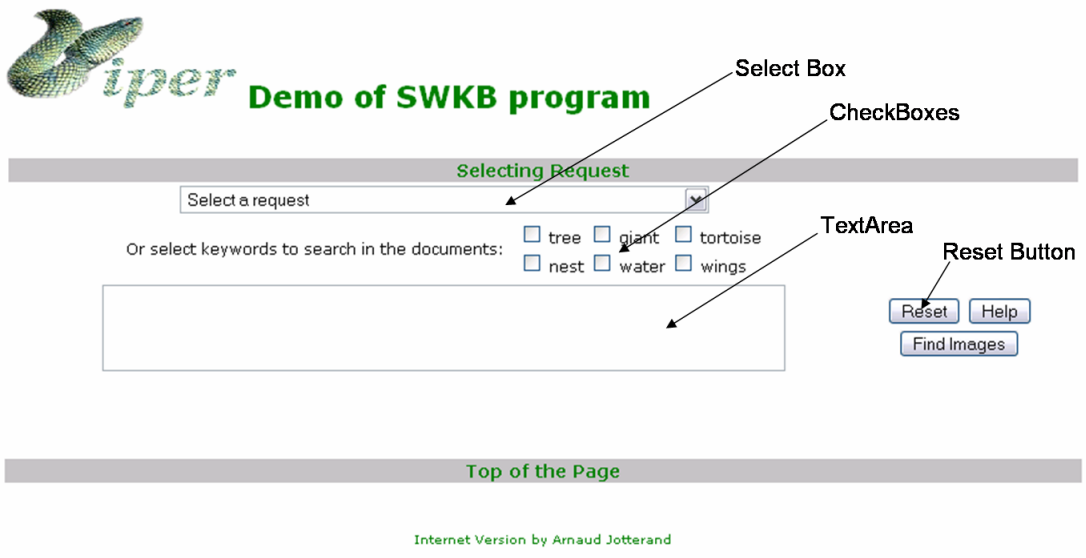

Figure 4.1. Basic situation of the webpage

4.1.2 Find Images

Once the query selected and/or edited, we are ready to execute the research for the corresponding images using the SWKB program; to do so, we just

have to click on the *Find Images* button. The results will appear below, and we can find (see figure 4.2) :

- The number of found images
- The images themselves

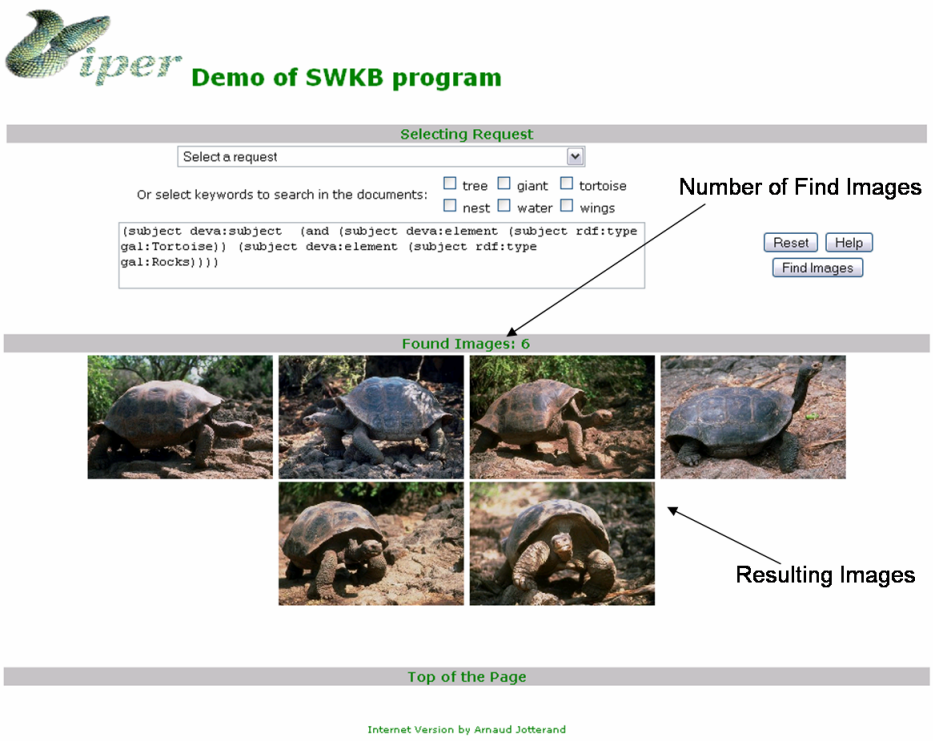

Figure 4.2. The webpage with the results part

4.1.3 Detail Image

All the resulting images are displayed in the results part of the webpage; it is possible to have many images, so there are just displayed without any detail. If we want to see the details of a given image, we just have to click on this image, and the details will appear in the details part of the webpage, situated between the query and results parts. We can find (see figure 4.3) :

- The image itself

- Its annotations in *.owl* file format

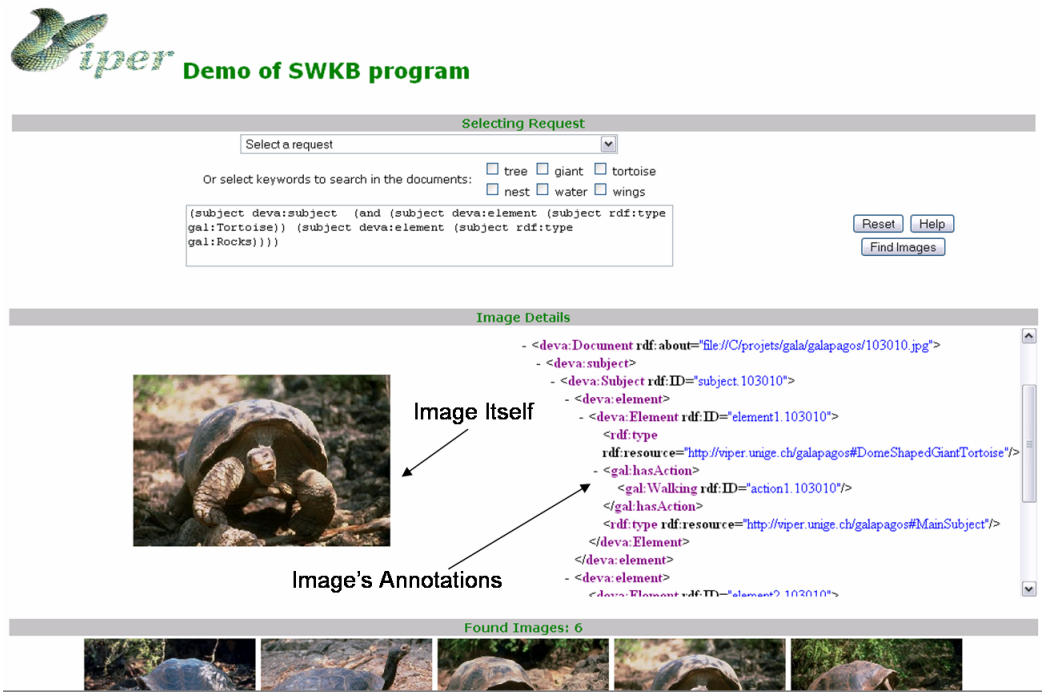

Figure 4.3. The webpage with the details part

4.2 Application's Other Pages

4.2.1 Details Page

Once an image is selected and displayed with its details in the details part of the main webpage, we can display these informations in a new window (for printing for example) simply by clicking on the image situated next to the annotations. We will have something like figure 4.4.

4.2.2 Error Page

Normally it should not happen, but it is possible that an error occurs. In this case, we will see a page similar to figure 4.5. From here, we can return

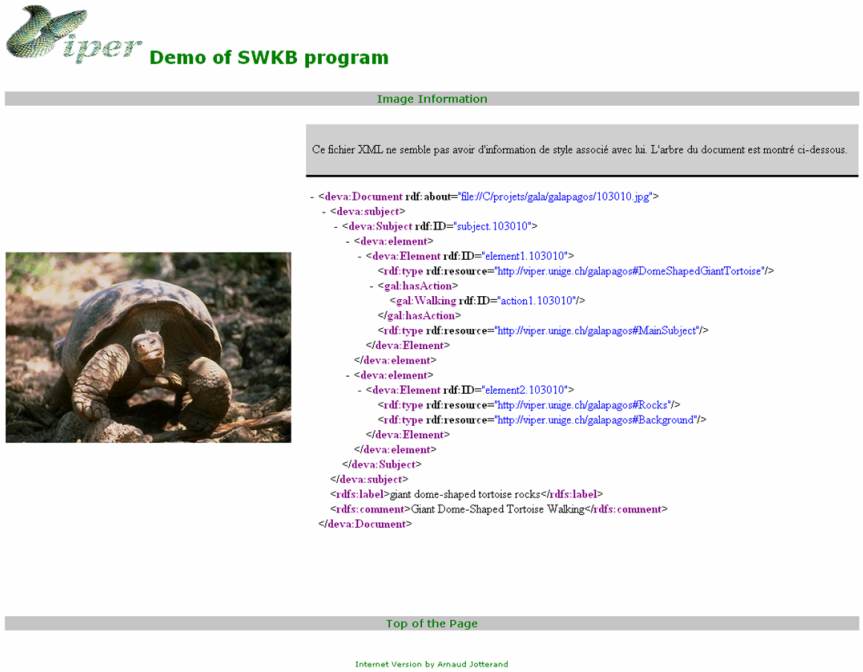

Figure 4.4. The details page of the application

to the main page by clicking the link envisaged for this purpose (see figure 4.5).

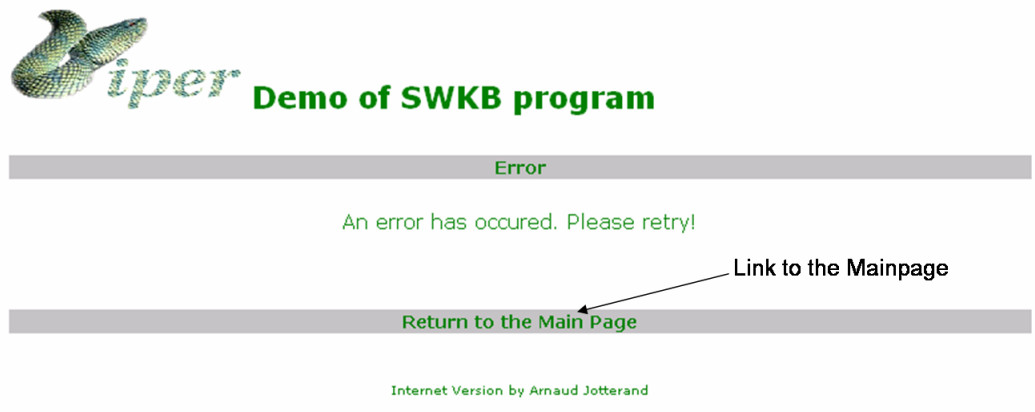

Figure 4.5. The error page of the application

4.2.3 Help Page

The use of this application is quite easy and intuitive. However, we can find help at any time simply by clicking the *Help* button situated in the query part; then a pop-up window similar to figure 4.6 will appear, containing all the information needed. This window can be closed simply by clicking the link envisaged for this purpose (see figure 4.6).

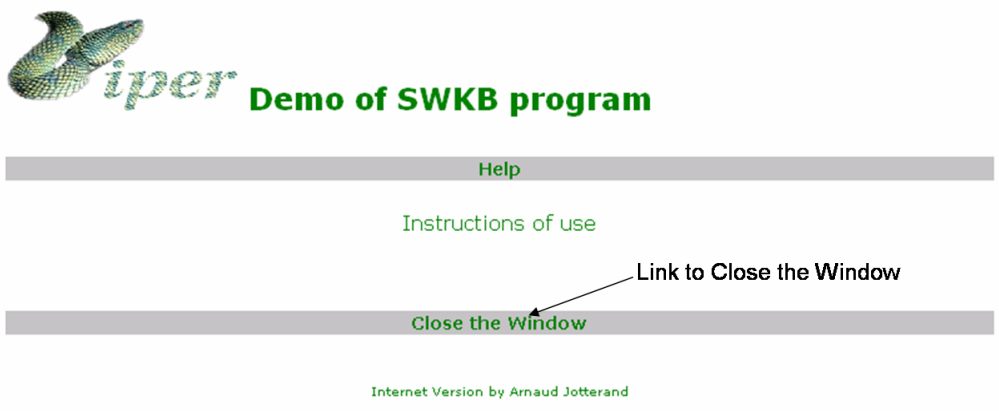

Figure 4.6. The help page of the application

4.2.4 Viper's Homepage

This application is a demonstration of a program developed by the Viper research group; it can be interesting or useful to visit the homepage of Viper. Thus, in every webpage of this application we can find a link to the Viper's Homepage; this link is the Viper logo situated in the top-left corner (see figure 4.7).

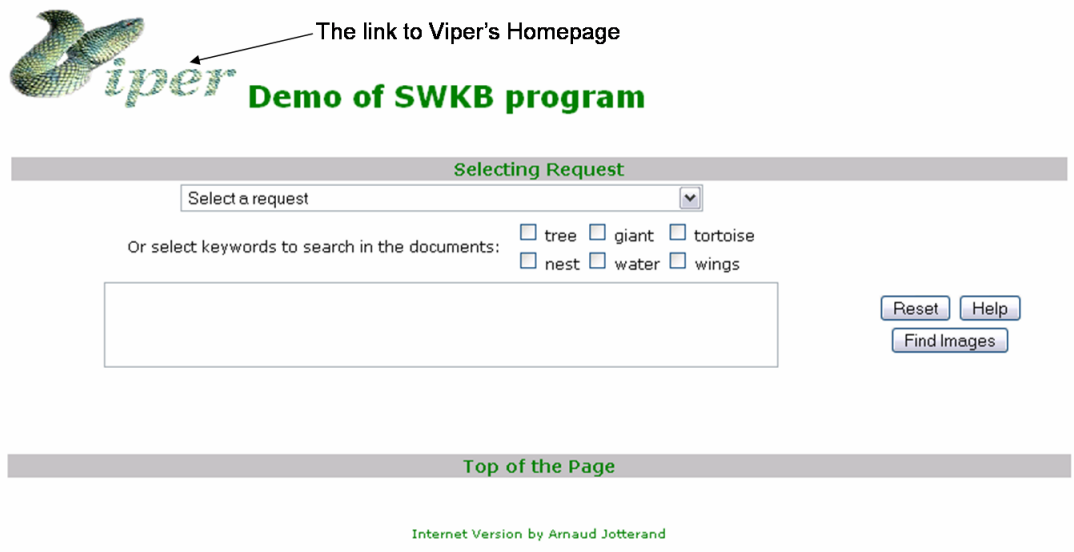

Figure 4.7. The link to the Viper's Homepage

Chapter 5

Conclusion

The goal of this project was to implement an internet version of a program developed by Viper research group. This goal has been reached, and even more, because the internet version contains some extra features in order to better show the power of this tool.

This project was also a good opportunity to use and be at ease with powerful technologies of Java 2 Platform, such as the JSP pages or the JavaBeans. It was also very interesting to convert a Java program to its internet version, because we can remark the advantages and disadvantages of each way of implementation; some things are easy to implement in one version, but can bring many problems in the other version.

The main encountered difficulties were in relation with the structure of the application rather than with the implementation itself. The necessary code was quite easy to write; there is no complicated algorithm to implement in such a web application. However, there were many little things to be aware of, such as the paths to the files or the place to put specific lines of code in the files; we can spend a lot of time searching for a bug in the code, whereas the problem is related to the location of the files.

In conclusion, I would like to personally thank my supervisor, Professor Stephane Marchand-Maillet, whom it was a pleasure to work with.

Appendix A

Acronyms

SWKB Semantic Web Knowledge Base

J2SE Java 2 Platform, Standard Edition

JSP JavaServer Pages

JSTL JSP Standard Tag Library

TLD Tag Library Descriptor

MVC Model-View-Controller

API Application Programming Interface

JAR Java Archive

CSS Cascading Style Sheets

HTTP HyperText Transfer Protocol

HTML HyperText Markup Language

XML Extensible Markup Language

SVG Scalable Vector Graphics

WML Wireless Markup Language

SQL Structured Query Language

Bibliography

[1] Stephanie Bodoff Debbie Bode Carson Ian Evans Dale Green Kim Haase Eric Jendrock Eric Armstrong, Jennifer Ball. The j2ee 1.4 tutorial. 2005.

[2] Marty Hall. Core servlets & java server pages. 2000.

[3] Javaserver pages standard tag library 1.1 tag reference. URL: `http://java.sun.com/products/jsp/jstl/1.1/docs/tlddocs/index.html`.

[4] Coding java servlets, includes java server pages. URL: `http://www.vipan.com/htdocs/usefulservlets.html`.

[5] Peter Harrison & Ian McFarland. Tomcat par la pratique. 2003.

[6] Hans Bergsten. Javaserver pages. 2001.

[7] Phil Hanna. Guide du developpeur: Java servlets et jsp. 2002.