Building Server-Side Web Language Processors

Ariel Ortiz
Information Technology Department
Tecnológico de Monterrey, Campus Estado de México
Atizapán de Zaragoza, Estado de México, Mexico. 52926
ariel.ortiz@itesm.mx

ABSTRACT

This paper discusses some useful insights for instructors who might want to consider using a web approach in courses involving language design and implementation. The basic idea is to have students build a language processor that actually runs on the web, instead of a processor that just runs on a command-line shell. The author documents the advantages and possible shortcomings of this approach, as well as his class experiences.

Categories and Subject Descriptors

K.3.2 [**Computers & Education**]: Computer & Information Science Education—*Computer Science Education*; D.3.4 [**Programming Languages**]: Processors—*Code Generation, Compilers, Interpreters*; C.2.4 [**Computer-communication Networks**]: Distributed Systems—*Client/server*.

General Terms

Design, Languages.

Keywords

Programming Languages, Compilers, Interpreters, Web Development, Web Frameworks.

1. INTRODUCTION

Compiler design courses have been a relevant part of the computer science undergraduate curricula at many universities around the world. Generally, students build a fully functional, yet relatively small, compiler during one semester. This compiler project commonly requires the translation of a subset of a conventional procedural language like C or Pascal. Using specialized tools, such as scanner and parser generators, students develop their translator going through all the well known compiler phases: lexical/syntax analysis, semantic analysis, intermediate code generation, code optimization, and object code generation. At the end, they have a standalone compiler that in general works just like the typical standalone compilers they have used in the past to build typical standalone applications.

In recent years, the software development efforts of many individuals and organizations have shifted from building standalone programs to building web-based applications. An important number of CS courses have been adapted to conform to this shift. For example, a CS1 course might have its students learn the Java programming language by building applets and running them in a web browser. Similar examples can be found in courses such as Operating Systems, Databases, Human-Computer Interactions, Computer Graphics and Software Development. Courses that deal with language design and implementation can also have a web orientation. In this case, students build a language processor that runs on the web.

This paper is a summary of the author's experience on teaching undergraduate students how to build interpreters and compilers that can be integrated into a web environment.

1.1 Why use a web-based approach?

Implementing a web language is not a trivial task. Here are two important reasons that justify this approach:

- **Curriculum organizing principles.**

  In the original CC2001 Computer Science Volume [2] Section 8.2, a number of principles for organizing the curriculum beyond the introductory courses was defined. Two specific approaches are worth mentioning here. To quote:

  > *A compressed approach:* For institutions that need to reduce the number of intermediate courses, the most straightforward approach is to combine individual topics into thematic courses that integrate material from related areas of computer science. . . . this strategy also begins to address the problem of classes that focus too narrowly on "software artifacts" . . .
  > *A web-based approach:* This model has grown out of a grass-root demand for curricular structures that focus more attention on the Internet and the World-Wide-Web, using these domains to serve as a common foundation for the curriculum as a whole.

  Thus, incorporating web technology and language implementation into a single course could be part of a hybrid curriculum, as described is Section 8.3 of CC2001, that may also meet more effectively the needs of a particular institution.

- **It is relevant and appealing to our students.**

  We can't ignore the fact that many of our CS students will become professional web developers once they graduate. Showing them how the web works from the perspective of a web language implementer gives them a deeper insight of many important technical issues.

  Additionally, as pointed out in [7]: "web programming promotes diversity in computer science because the programs are more motivating to a diverse audience". The author has witnessed that students have a much more engaging learning experience when they develop a web language than when they just build a conventional language translator. This is probably because it is more exciting to see a program running on a web browser than on a command-line shell.

1.2 General Concepts

1.2.1 Language Processors

A *programming language processor* is "... any system that manipulates programs expressed in some particular programming language. Language processors allow us to run programs, or prepare them to be run" [12]. For this discussion, a language processor can be a compiler or an interpreter, or even a combination of both known as an *interpretive compiler*, like those used in most Java implementations.

1.2.2 Runs on the Web

The World-Wide Web is a *client-server* architecture. This means that there are two different physical locations in which a web language processor can run:

Client-side language processors: These run within a user-agent, most commonly a web browser. Possible implementation technologies include: ECMAScript (JavaScript), Java applets, Flash, Silverlight, JavaFX, and plugins for specific browsers.

Server-side language processors: These run on the server computer and somehow interact with a web server. Almost any modern programming environment can be used to develop these types of processors. Section 3 covers three possible implementations.

The remainder of this paper will only focus on server-side technologies.

2. DESIGN CONSIDERATIONS

What specific characteristics must a web language have? Actually, it can be similar to any conventional programming language. However, it is probably best to design a web language so that it has at least a few "scripting" features (for example: compact syntax, dynamic types, garbage collection, and direct support for high level collections such as strings, lists and dictionaries) if one wishes to make it more like the current languages used for web development (for example: JavaScript, PHP, Perl, Ruby or Python). Anyhow, all web specific features can be made available to almost any kind of language through runtime library functions.

The next subsections point out some of the issues worth considering during the design of a web language.

2.1 Overall Syntax

The purpose of a web language is to do some computations and then produce an output in the form of an HTML web page (or XML document, plain text, etc.). This means that a resulting page is built from two types of elements: *dynamically generated elements* and *static content elements.* The overall syntax of the web language should reflect this. There are two possible ways in which these elements can be combined within one source program:

- The presentation code (static elements) can be embedded inside the web language code (dynamic elements), for example:

  ```
  print "<html><body><ul>"
  for i in 0..10 do
    print "<li>" + (2 ** i) + "</li>"
  end
  print "</ul></body></html>"
  ```

  In this example, the `print` statement is used to generate the desired output.

- The web language code can be embedded inside the presentation code. This is known as a *Template View* [6] and it is the preferred notation because of its similarity with the desired output. An example:

  ```
  <html><body><ul>
      {% for i in 0..10 do %}
        <li> {{ (2 ** i) }} </li>
      {% end %}
  </ul></body></html>
  ```

  The delimiters `{{ ... }}` indicate that the surrounded expression should be evaluated and its result displayed as part of the output. The other delimiters `{% ... %}` represent statements that are to be executed for their effect, and don't produce any output by themselves. Everything outside these delimiters is static content, so it is part of the output.

For implementations purposes, the language processor might receive a program like the one in the second example, transform it to something like the first example, and then proceed to execute it. This is actually how JavaServer Pages (JSP) [3] work.

2.2 Hypertext Transfer Protocol

A general understanding of the Hypertext Transfer Protocol (HTTP) [8] is fundamental when designing and implementing a web language. HTTP is a request/response standard between a client and a server. An HTTP client initiates a request by establishing a Transmission Control Protocol (TCP) connection to a particular port on a host computer. An HTTP server listening on that port waits for the client to send a request message. Upon receiving the request, the server sends back a response. This response can be the contents of a static file (like an image or an HTML page), or it can be a dynamically generated resource produced by a web language processor.

2.3 Names and Scoping Rules

Web language applications should have all the request headers and parameters readily available. One option is to provide a library of functions, like `get_header("name")`. Another option is to use a special naming convention. So, for example, if a variable has a dollar symbol prefix, it can represent a request parameter, an exclamation mark prefix could represent a header, and so on. Dealing with the response can follow similar approaches.

It is also important to give some thought on how scoping rules should work. A name with *local scope* can only be used within the procedure that defines it. Alternatively, a name can have a *page scope* (visible in the whole web page or document), *request scope* (visible through several pages chained by the same request), *session scope* (visible throughout the user session), or *application scope* (visible to all the application users). Library functions or naming conventions can also be used to manage names in different scopes.

2.4 Security Issues

A web language should provide a simple mechanism for filtering special characters before sending them as output. If a user input contains the characters `&`, `<`, or `>`, these should be probably replaced by `&amp;`, `&lt;`, or `&gt;` when sending them back as output to HTML and XML documents. Not doing so makes a web applications susceptible to HTML Injection and Cross-Site Scripting (XSS) exploits [11]. Emphasizing these kinds of issues helps promoting security awareness among our students.

A special function can be used when the result of an expression should be filtered within a web program, for example: `{{ filter("some <output>") }}`. Better yet, the default behavior of the pair of braces `{{` ... `}}` could automatically do any necessary filtering, while a different pair of delimiters `{!` ... `!}` could be used when no filtering should be done.

3. IMPLEMENTATION STRATEGIES

To keep things as simple as possible, most of the development and testing of any web language processor should be carried out on a command-line shell, using stubs if required. The integration with a web technology should be attempted only when the main functionality of the language processor is already in place.

For the sake of keeping the following discussions short, the implementation details will assume that the web language processor being developed is actually an interpreter instead of a compiler. Hopefully, the knowledgeable reader should not have any major problem extrapolating the information presented here in order to use it in the context of a compiler or an interpretive compiler.

3.1 Using the Common Gateway Interface

The Common Gateway Interface (CGI) is a simple interface for running external programs under a web server in a platform-independent manner [9]. When a client sends a request, the web server determines if the requested resource is a file stored on disk or an executable program. In the former case, the web server takes the file, pre-appends it with the status line and headers, and sends that as the response to the client. In the later case, the web server runs the executable program as a new process, and whatever output it produces is sent to the client as the response.

Popular scanner and parser generator tools (like *lex* and *yacc*) can be used to build a CGI-based language processor which can be invoked by a web server, such as the Apache HTTP server [1].

Any programming language can be used as an implementation language for a CGI-based web language processor, as long as it is able to:

1. Write to the standard output (to generate the response).
2. Read from the standard input (to read the body of the request).
3. Read OS environment variables (to read the request headers).

The response headers and body are generated by writing the desired information to the standard output. The web server completes the response message by adding the status line and some server specific headers to the start of the CGI output, and sends the entire result to the client.

CGI was the first technique available for generating dynamic content in the World-Wide-Web, dating back to 1993. Its main drawback is that the web server has to create a new process for every request involving a CGI. Process creation is an expensive operation in most operating systems, thus the server performance can be severely affected if many requests arrive during a short time period. This could be a common scenario in a production environment, but in a student project this is probably never the case.

CGI seems an old and low-tech approach, especially when compared to the web development technologies available today. It is important to consider this, because students already familiar with other web tools and techniques might be reluctant to use CGIs. Yet the approach is straightforward, and it allows using all the language construction tools we are all familiar with. CGIs are at least worth considering.

3.2 Building Over an Existing Web Technology

When developing a web language processor, why not use all the infrastructure already available from some existing web technology? For example, JavaServer Pages allow defining custom tags [3]. The body of a custom tag can be any text, including the source code for an arbitrary programming language. Thus, it is possible to have a JSP source file like this one:

```
<%@ taglib prefix="lispish"
           uri="LispishCodeTagLibrary" %>
<html><body>
  <lispish:stm>
    (define fact
      (lambda (n)
        (if (= n 0) 1 (* n (fact (- n 1))))))
  </lispish:stm>
  Factorial of 5 is
  <lispish:exp> (fact 5) </lispish:exp>
</body></html>
```

In the previous example, the `taglib` directive allows importing a tag library referred as `LispishCodeTagLibrary`. All the tags for this library will use "`lispish`" as their prefix.

The `<lispish:stm>` tag executes its body for its side-effects only, while the `<lispish:exp>` tag evaluates its body and writes the result to the output.

These are the general steps to write a JSP custom tag:

1. A Tag Library Descriptor (TLD) file must be created. This is an XML file that contains information about the library as a whole and about each tag contained in the library. The tag declaration inside the TLD file must specify that its body content is `tagdependent` so that the web container (for example Apache Tomcat) treats the tag's body as plain text with no special meaning.

2. A Java class has to be defined, and it must extend the `javax.servlet.jsp.tagext.SimpleTagSupport` class. The class must also contain an overridden `doTag()` method. This method is called automatically by the web container when the tag is to be processed at request time. Inside this method it is possible to obtain the body contents of the tag. At this point, the web language processor can be called to carry out its duty.

The main advantage of using an approach like this one is that the web language processor has access to the all the facilities provided by the hosting environment. Things like session tracking, cookie management, request parameter parsing, header processing, and response generation are typically just a library method call away. On the other hand, the main downside is that you might end up having less options when selecting tools to build a language processor due to the restrictions imposed by the hosting environment. This means that if you want to build a language processor that is available as a JSP custom tag, you can only use tools available for the Java platform.

3.3 Programming Your Own Web Server

This implementation strategy consists of programming a web server from scratch. The web server should be able to serve static resources, but its main function is to host a language processor that allows generating dynamic content.

The server component can be implemented using any language supporting TCP sockets. The language processor should most likely be implemented using that same language.

The following general outline describes how a simple web server with an embedded language processor could work. Assume SS is a server socket. Repeat the following steps indefinitely:

1. Listen for a connection to be made to SS and accept it. Let S be the new connection socket.

2. Serve the request received through S using an available or newly created thread:

   (a) If the resource requested is a static resource Q, map the file extension of Q to its corresponding MIME type so that the correct `Content-Type` header is reported. Add also to the response message the contents of Q plus a 200 status code and send it back through S.

   (b) If the resource requested refers to a source program P, execute P using the corresponding language processor and let O be its output. Build the response message with O plus a 200 status code and send it back through S. If P produces a runtime error, the response should instead be: `500 Internal Server Error`; send it back through S.

   (c) If the resource requested does not exist, the response should be: `404 Not Found`; send it back through S.

The main disadvantage of this approach is that practically everything in the HTTP protocol has to be explicitly dealt with: headers have to be parsed, request parameters have to be decoded, errors have to be detected and reported, and so on. On the bright side, implementing a web server gives a tremendous insight on the low-level details of web development.

Having students write a web server from scratch can be deemed as too complex and/or time consuming. A simple way to alleviate this chore is to give students the source code for a simple web server kernel and allow them to extend it in order to incorporate new functionality. Some languages might even have a standard (or third party) API offering some support for building web servers. For example, Python includes the module `SimpleHTTPServer` that provides a basic HTTP request handler mechanism [10].

4. CLASS EXPERIENCE

Since 1998, the author has been sporadically using the described implementation strategies of Section 3 to teach three different courses at the Tecnológico de Monterrey, Campus Estado de México. These courses are part of a nine semester Computer Systems Engineering undergraduate program, which basically blends together computer science with software engineering. The following subsections briefly describe representative project examples for each of these courses.

4.1 Programming Languages

Programming Languages is a 5^{th} semester course focused mainly on teaching different programming styles (functional, logic, concurrent, etc.). It concludes with a brief introduction to language translators. Course prerequisites include Data Structures and Theory of Computation.

Students were once required to write an infix expression evaluator using *flex* and *bison* with C. The evaluator was available through a CGI as described in subsection 3.1. A user interface was provided as an HTML web form containing a text field where the expressions could be typed. When the form got submitted, the CGI was called, and the result was displayed in a new HTML page.

This project was developed in teams of two people, with a time requirement of about two weeks. Almost all of the projects worked flawlessly.

4.2 Language Translators

This is a conventional Compiler Construction course, offered during the 8^{th} semester. It has Programming Languages as its prerequisite.

In one particular semester, the project consisted in writing a compiler that was available as a JSP tag, just like the one explained in subsection 3.2. The compiler was written in Java using the JavaCC scanner and parser generator. It produced Java Virtual Machine (JVM) assembly language code, which was immediately assembled into byte-code. The byte-code was then dynamically loaded and executed by the

JVM, which was also running the web container. All this process was actually orchestrated during a single client request.

This project was developed during a whole semester. Students were allowed to work individually or in teams of two people. The code generation phase as well as the integration into the web container proved to be quite challenging to an important number of students. At the end, about two thirds of the projects worked as expected.

4.3 Software Development Project

This is a 6th semester course that focuses on the integration of the knowledge and skills acquired during the second third of the program through the development of a software-based application. Its prerequisites are: Analysis and Modeling of Software Systems, and Web Applications Development. The project developed during the course can have two orientations: as a computer-science related project (software systems with a specific goal such as translating, searching engines, machine controlling, etc.) or as a Business Information System. The course was introduced into the Computer Systems Engineering program in 2008. At the same time, the Language Translators course was removed.

On one occasion, students were required to develop an MVC (Model-View-Controller) web framework using the Erlang programming language [5]. The view component was developed using *yecc* (an Erlang parser generator) and it was basically a template system inspired by the Django framework [4]. The support for the controller and model components was implemented directly in Erlang. Students also developed their own web server as explained in Section 3.3, taking advantage of the concurrency facilities available in Erlang.

For this full semester project, students worked in teams of three or four members using the Extreme Programming (XP) development methodology. Iterations lasted one week and were closely supervised by the instructor.

All the projects were successfully concluded. Students' feedback was extremely positive. Here are some quotes taken from student surveys:

> In this course I learned how programming languages and compilers work, just like those I've been using since my first semester. It was a good experience as a future Computer Systems Engineer, specially because my program's curricula does no longer have a Compilers course. ...I also learned something vital: the inner workings of the MVC architecture, which is employed in many applications. There's no doubt that I learned a lot thanks to this project.

Another student wrote:

> What I most liked about this project in particular (an MVC framework) was that I was finally able to blend together the things I had learned previously in college. We built an application that allowed us to build other applications! ...

And a last quote:

> ...Definitely, this course exceeded my expectations. It was a course in which I was able to relieve my passion for software development.

5. CONCLUSIONS

It is possible to incorporate successfully a web-based approach into courses involving language design and implementation. For the author, the best results were obtained in the Software Development Project, but it is important to note that other factors, such as the use of XP, also influenced the outcome.

When using a web-based approach in more traditional courses, such as Programming Languages and Compiler Construction, the main drawback experienced by the author is that additional material needs to be presented to students. This material can be lectured in as little as one or two weeks out of a 16 week semester. It is more work for the students and the instructor, but the results are very rewarding.

Complete source code examples for the implementation strategies of Section 3 are freely available for any one to use and modify at: `http://weblang.arielortiz.com/`

6. REFERENCES

[1] The Apache Software Foundation. *The Apache HTTP Server Project.* http://httpd.apache.org/ Accessed November 7, 2009.

[2] CC2001. *Computing Curricula 2001: Computer Science Volume.* ACM/IEEE Joint Task Force on Computing Curricula, 2001.

[3] K. Chung and J. Luehe. *JSR 245: JavaServer Pages Specification, Version 2.1.* Sun Microsystems, 2009. http://jcp.org/en/jsr/detail?id=245 Accessed November 7, 2009.

[4] Django Software Foundation. *The Django Framework.* http://www.djangoproject.com/ Accessed November 7, 2009.

[5] Ericsson AB. *Open Source Erlang.* http://www.erlang.org/ Accessed November 7, 2009.

[6] M. Fowler, D. Rice, M. Foemmel, E. Hieatt, R. Mee, and R. Stafford. *Patterns of Enterprise Application Architecture.* Addison-Wesley Professional, 2002.

[7] J. Miller. (jessica.k.miller@gmail.com) (March 26, 2009). *SIGCSE 2009 Web Programming Recap and Welcome to New Mailing List Members.* Email to: Webdev mailing list (webdev@cs.washington.edu).

[8] Network Working Group. *RFC 2616: Hypertext Transfer Protocol – HTTP/1.1*, 1999. http://tools.ietf.org/html/rfc2616 Accessed November 7, 2009.

[9] Network Working Group. *RFC 3875: The Common Gateway Interface (CGI) Version 1.1*, 2004. http://tools.ietf.org/html/rfc3875 Accessed November 7, 2009.

[10] Python Software Foundation. *Simple HTTP request handler*, 2009. http://docs.python.org/library/simplehttpserver.html Accessed November 7, 2009.

[11] J. Rafail. *Cross-Site Scripting Vulnerabilities.* CERT Coordination Center, Carnegie Mellon University, 2001.

[12] D. Watt and D. Brown. *Programming Language Processors in Java: Compilers and Interpreters.* Prentice Hall, Harlow, England, 2000.