DIGUA: MINIFIER AND OBFUSCATOR FOR WEB RESOURCES

Alex Ciminian and Ciprian Dobre
Department of Computer Science
Politehnica University of Bucharest
Spl. Independentei, 313
Bucharest, Romania
E-mail:
alex.ciminian@gmail.com
ciprian.dobre@cs.pub.ro

KEYWORDS

Digua, minification, web, front-end, javascript, css.

ABSTRACT

This paper presents Digua, a standalone library for minifying web resources. It is designed to be used for reducing file sizes of CSS, JavaScript and HTML files. A side-effect of minifying these resources is code obfuscation.

The novel approach in our solution is the fact that it can minify all elements of a website, and not just one type of resource. Also, it can detect the correlation between HTML, JavaScript and CSS and perform an all-round minification, therefore saving more bytes. There are no similar solutions available on the market today, technology-wise and functionality-wise. The main challenge is to preserve the code's functionality. It is hard to ensure that this type of operation will function across an almost infinite variety of websites, but we have taken steps to mitigate these risks through a flexible benchmark system.

INTRODUCTION

The speed of the Internet has become the focus-point of many companies today. Serving content and applications as efficiently as possible is a challenge most developers have to face at the present moment. Faster load times usually contribute to a better overall user-experience that in turn leads to more content consumption online. This generates more revenue for content publishers as well as for advertisers so it is no wonder that big Internet companies are investing heavily in both creating and promoting tools that promise a faster online experience.

Our project is tied to the developer space of Internet speed. More precisely, we address optimizations in the front-end, because it has been estimated by analyzing HTTP traffic (Souders, S 2008) that for most web applications the user waits 80% of the time for the page's resources to be loaded. The other 20% is spent waiting for backend computation and for the html to be served.

We propose a solution that reduces file sizes in order to make web pages load faster. We do this by leveraging industry best practices (Souders, S 2007) (Souders, S 2009) and offering users the possibility of applying these practices automatically to their projects. The solution is production-ready and can be used in multiple ways: through the command line, integrated in build environments (e.g. Ant, Maven). Further developments will make usable through a GUI or plugged into an application server as a filter. One of the project's objectives is to make it as easy as possible to integrate the minifier in a variety of workflows.

STATE OF THE ART

There are multiple techiques that have the goal of reducing a website's load time and bandwidth usage. Minification is one of these; it is the process of removing all unnecessary characters from source code, without changing its functionality. These unnecessary characters usually include white space characters, new line characters, comments, and sometimes block delimiters, which are used to add readability to the code but are not required for it to execute.

Minification can be distinguished from the more general concept of data compression in that the minified source can be interpreted immediately without the need for an uncompression step: the same interpreter can work with both the original as well as with the minified source.

The reference implementations we compared our solution to are the YUI Compressor [Yah08], the Google Closure Compiler [Goo10], Dojo ShrinkSafe [Too07], Packer [Edw07], JSMin [Cro03] and CSSMin [Scy04]. These are the most widely used tools in the industry today.

Among these, the YUI Compressor and Google Closure set of tools are the only ones that provide a wider range of options and supported formats when minifying web resources. Other tools focus on single file formats and are less configurable.

Packer is the most atypical tool from the ones analyzed. It is based on encoding the source JavaScript in Base62, a positional notation [Wik11] that yields very good compression results. However, this does come at a cost - there is time spent on the client 'unpacking' the original source code, and sometimes it may be more than the time saved downloading the file! This is highly dependent on the type of hits the application is expected to receive. If visitors are expected to return often, packer is disadvantageous because of the extra CPU load it incurs on decompression.

ARCHITECTURE OF THE SOLUTION

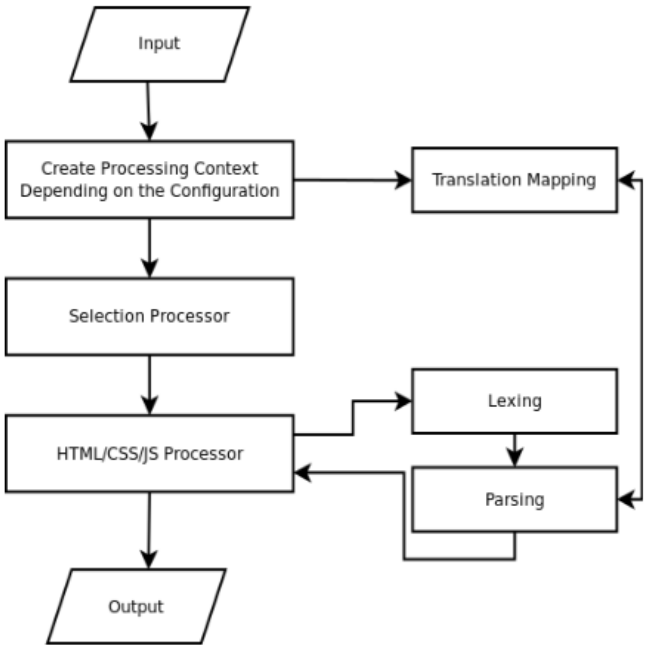

Figure 1: Program Flow

Input and output

The input and output operations are abstracted through a set of classes that implement the `Source` interface in our project. They are flexible and genericized, thus make Digua usable in a variety of contexts. The project currently supports `File`, `Directory` and `String Sources` (for direct text input). There are plans to implement `ServletRequestSources` so that the library can be plugged into an application server as a Filter (as specified in the Java Servlet API reference [Ora11]).

Context

The context holds references to the input and output sources for a transformation and the map of translated variable names. It also can be passed a reference to a parent context from which a common map of translated variables can be read and the processors can be kept in sync. The context also holds references to two Sources that determine how the input and output are handled. A context must be passed in each processor.

Lexers and Parsers

The lexers and parsers are automatically generated from our grammars via ANTLR. While the Lexer is left as is, the parsers are heavily modified and contain the logic that does the actual minification of the code. All Parsers inherit from the DiguaParser abstract class which makes the Context available inside inheriting parsers and contains the common token filtering and output logic.

The minification rules are passed inside the grammars. For example, let's take the case of minifying HTML colors. In CSS, the colors can be specified in hex format or, for a few of them, as plain English color names. To make all paragraphs on a page red, one could have a rule like `p { color: #ff0000; }`. The hex color can be written in condensed form as `#f00`, or even better, as `red`. This simple optimization can save up to 4 bytes for each declaration. An example on how this is achieved can be seen in this gist[Cim12].

Processors

The processors are the classes doing the core work. They take the input, process it, and output the results. The source and destination are determined through the context being passed into the process method specified by the processor interface.

For processors that use lexing and parsing, the lexer and parser are referenced internally, through their respective classes. The processors may host content specific methods (for example, function translation is specific to JavaScript) that canbe used in the parser and lexer (the processor is itself passed as a property for those classes). The abstract ParserProcessor is extended by all classes that use parsing. It instantiates the respective lexer and parser classes at runtime, so there is no need to duplicate the interface method process in the specific classes that extend it (`HTMLProcessor`, `JavaScriptProcessor`, `CSSProcessor`).

We also implemented several "utility" processors. The `SelectionProcessor` can choose the right type of processor based on the content type of the input. The `SerialProcessor` can be used to chain processor calls, for example to output minified CSS directly from a LESS preprocessor file. The `ParallelProcessor` can be used to speedup the minification time for multiple files by processing them in separate threads. Figure 2 shows the tree structure of the processors.

Translators

The translators determine the names of the variables, ids, functions etc. after minification. There are several strategies for translating variable names and are all configurable. Digua can translate only local variable names and not modify the global ones. It also can generate random variable names in the minified version thus obfuscating the source. Alternatively, it has the option of generating sequenced variable names, which make the source, if unminified, easier to understand.

The translation of names is being kept consistent by a shared map, per processor. For example, in the case of processing a full directory, there would be several Translators invoked (for JavaScript: a translator for local variables, one for global variables, one for function names) but only one translations map which would be held in the parent context of the application.

Technologies Used

The main part of the project is written in Java. To generate

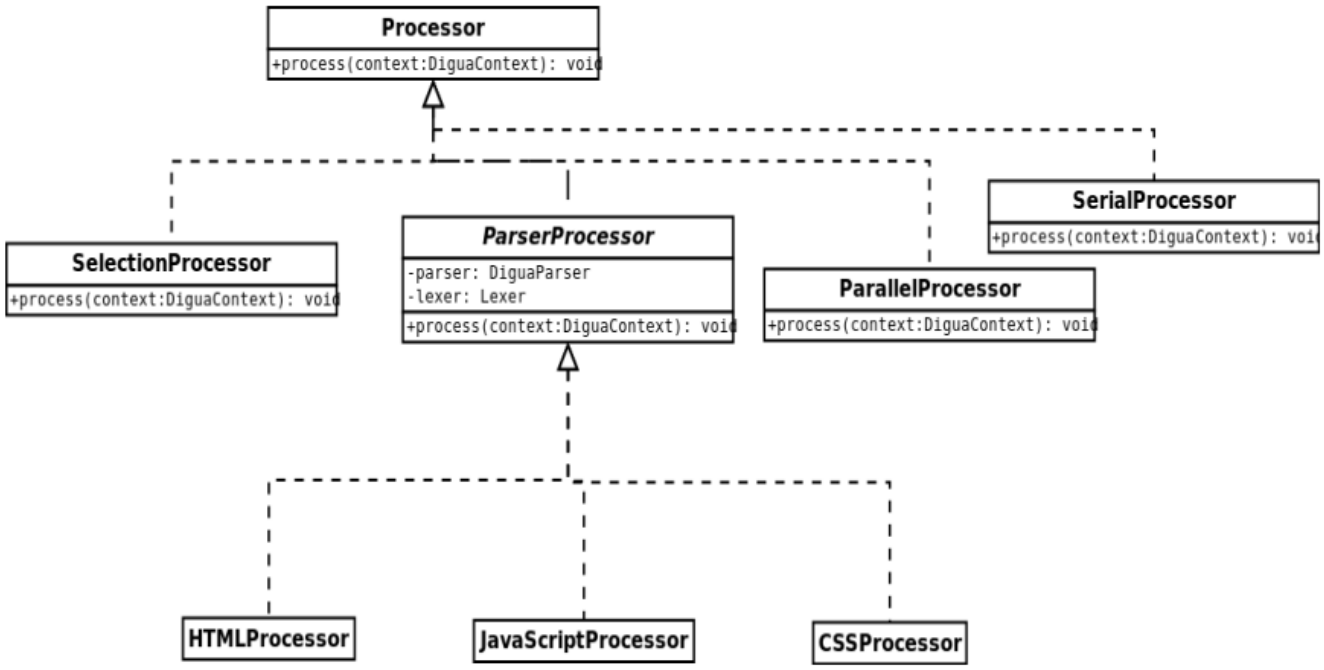

Figure 2: Processors Diagram

the parsers and lexers we used the ANTLR library. The available grammars for HTML, CSS and JavaScript were tuned for the scope of our minifier.

We implemented the project's build system with Apache Maven. This allows for automatic dependency resolution and enables the project to be integrated easily in modern builds. Also, by using Maven we provide seamless integration with the Eclipse IDE, since you can generate the project files from the POM.

The benchmarking is done by running a shell script that in turn calls a python script configured through JSON. We used git to retrieve the actual tag for the source code we compressed and validated, but we also ran tests on custom HTML, CSS and JavaScript.

Contributing to the Project

All the code associated with this project is open-source (released under the Apache 2.0 License) and is hosted on SourceForge. By having the project on SourceForge [BC11] we also make use of their hosted apps like MediaWiki as documentation, Mantis as a bugtracker, code versioning via SVN and a presentation website for the project.

CASE STUDIES

There are two different directions we approached when evaluating our solution. The first one was comparative: we wanted to see how Digua fared against the top open-source libraries that have the same objective. The second one was absolute: we wanted to see what kind of performance improvement a user would hope to have by using our solution on a complete webpage. This second approach could not be included as a comparative test, since Digua is the only library offering this functionality.

We compared our solution against the libraries we mentioned in section 2. None of them provide the full functionality that Digua offers, but we crafted our tests around the features that our competitors support.

The benchmarking was done using a python script (open source, available in the project's repository). The script is externally configurable through a JSON file; new tests or new frameworks for the comparison may be added just by modifying that file.

Compressing jQuery

The first test features the reduction in file size obtained by each library when run against the latest stable jQuery source code file. The file was initially 161.5 KB in size. The test results are depicted in figure 3.

Dean Edwards' packer library fares best when compressing jQuery. This is because packer actually encodes the source JavaScript in Base62 and does not perform the tokenization that all other libraries, including Digua, perform. The tradeoff between the reduction in size and cost in computation time can prove to be a drawback, especially if you fewer fresh visitors than returning ones.

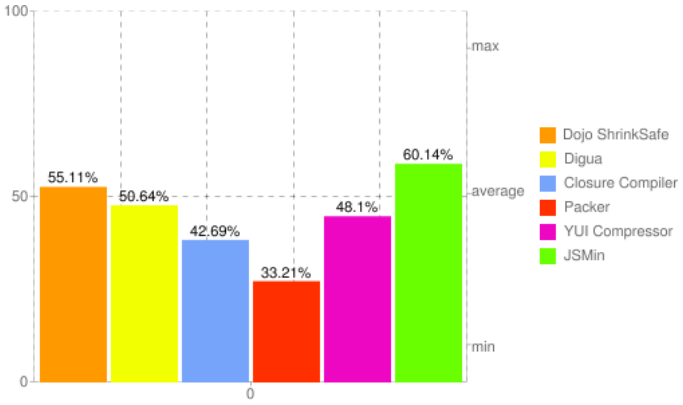

Figure 3: jQuery minification comparison

There were no validation tests run for the jQuery minification because of a bug in the way the tests on the library are run. In short, the jQuery project is made up of multiple source files

that get 'built' into the resulting framework file. There is currently no way to run the tests on a single-file source (whether it be minified or in its initial state), although the test suite claims to work in this situation. This is a bug we reported to the jQuery core dev team and will probably be fixed in future releases.

Compressing prototype.js

This test was similar to the one performed on jQuery, the only difference is that it was run on the Prototype JavaScript framework. The original source file measured 126.7 KB. The same procedure was followed: checkout the latest stable tag from source control, build the source and run the minifiers against it.

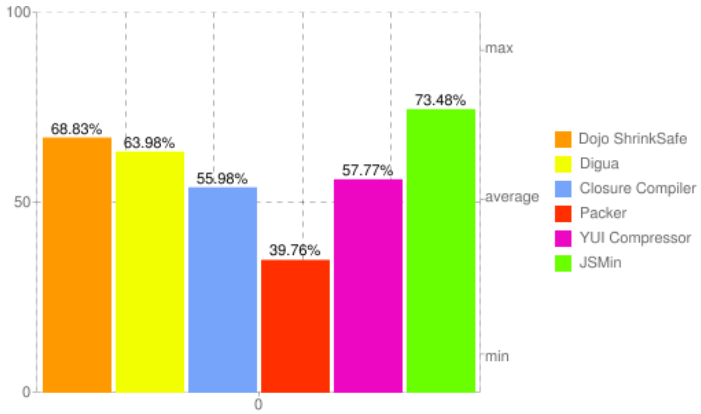

Figure 4: Minifying prototype.js

The main difference is that the prototype test suite could be run agains the minified source, so we were able to validate our result as being correct. This is important moving forward, especially for the introduction of new features.

Compressing a Wordpress Theme's Stylesheet

We tested CSS compression against the only two other libraries in our benchmark that offer this functionality (YUI and CSSMin).

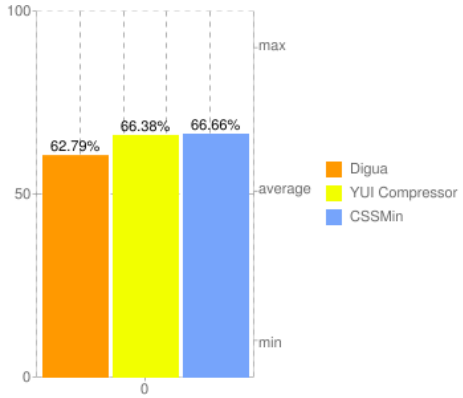

Figure 5: Minifying a Wordpress Theme Stylesheet

The results listed in Figure 5 show that there is not much room for improvement here, all libraries fared more or less the same. Also, there was no point in trying to validate the compression as it does not make much sense without and HTML context file on which to apply the styles. The target for compression was a rather large (16 KB) stylesheet file for the Glassical Wordpress CSS theme.

Full Webpage Compression Tests

We ran two benchmarks for full webpage compression. In the first scenario we tried to think of a typical website that loads the minified jQuery source but also tries to load several unminified plugins. No minification was assumed on the part of stylesheets and HTML. The results are promissing (Figure 6: the overall reduction in size is of almost 35%. The intial size was 107.7 KB. This is significant, and considering there is almost no difficulty involved it is by far worth the effort.

The second benchmark was for a website that already minifies its external resources, but does not run a solution over all its components. We took the Smashing Magazine homepage as an example - they are a leading resource for web developers and have extensive coverage of best practice techniques. We saved the webpage locally and ran digua over all the downloaded resources. This resulted in an almost 5% drop in overall size (we excluded the images from this calculation, as they were not processed in any way). Although, in the context of serving compressed resources this may not amount to much it may be significant for websites that have millions of pageviews per month.

CONCLUSIONS

The architecture of the library is easily extensible. New input and output sources can be added so that it can fit in any context, wether it be inside an application server, in the command line, in a graphical interface or in a web environment. To improve processing, hooks can be added in the grammar and functions may be implemented in the Processor classes (as described in the color minification example). To provide better obfuscation, additional translators may be implemented that yield even harder to understand code than the Random one present in the project. The minification is designed out to be configurable, although the interface for configuration is not currently exposed to the user.

The benchmarking we implemented will prove to be very valuable in the long run because it is very easily extensible - new tests can be added just by modifying the configuration file. The automatic reporting provides easy to read results in a matter of seconds (almost all the charts in this paper were generated by the benchmarking script).

The comparative tests will help us improve Digua by showing us which libraries do a better job of compressing certain resources thus pointing out certain features we can implement so that it will yield better results. The available automated validation is going to help in implementing these new features, because it can quickly confirm that they do not break the minification process and will speed up development time.

While it still lags behind the industry leaders (Google, Yahoo!, packer) when it comes to minifying JavaScript,

Digua's overall performance makes it a good choice as a minification tool. As speed is beginning to be seen as a feature, minification is being implemented on a larger and larger scale. Digua's versatility when it comes to the way it can be executed we hope will determine people to use it.

The project is open-source and will remain that way. A large part of this document, after it is processed, will be used as documentation for developers wanting to contribute to the project. We hope to attract as many interested people as possible because that is the best way to make the project grow and to improve its performance as much as possible.

FUTURE WORK

One important feature that currently the project is lacking would be to implement an external way of configuring the minification process. This would be available to the end-user without forcing him to modify the library's source code in any way. The code is designed well in this respect, but there is still a lot of logic to be implemented. For example, if one was to specify a JavaScript file as an input to Digua and that file contained a comment in the form:

```
/*@Digua.include ( 'other.js') */
/*@Digua.transform.variables.global=false */
```

It should also trigger the minification of the other.js file, but without transforming global variable names.

Integration with CSS preprocessors

Another feature that is currently lacking is integration with CSS preprocessors, such as LESS or SASS. These could be used with the SerialProcessor and would allow seamless integration for developers that rely on these tools to keep their stylesheets maintainable.

This is a feature that has not been implemented by any other library among the ones which we studied, although Google's Closure tool suite offers a CSS preprocessor of its own.

Graphical User Interface

The previous version of Digua also contained a graphical user interface, but the architectural changes in the present version have made it incompatible with the project's current state. A future development would be rewriting the GUI code. Although initially written to function with Swing, using SWT would make it easy to integrate with the Eclipse IDE. Distributing it as an IDE plugin would make it more attractive to developers and could increase the project's adoption.

Performance Enhancements

Implementing new features for minification may prove to further reduce file sizes. The groundwork is laid out for this, including benchmarking, validation and easy way to develop features so we hope to further increase the performance of the project in the near future.

ACKNOWLEDGEMENTS

A special thank you goes out to **Adrian Ber** who started this project in 2009 and wrote the first two versions completely on his own. His support was essential for the code contributions made in this new version and in the writing of this paper.

The research presented in this paper is also supported by the national project: "TRANSYS Models and Techniques for Traffic Optimizing in Urban Environments", Contract No. 4/28.07.2010, Project CNCSIS-PN-II-RU-PD ID: 238.

REFERENCES

Souders, Steve. 2007.
High Performance Web Sites. O'Reilly Publishing, CA.

Souders, Steve. 2009.
Even Faster Web Sites. O'Reilly Publishing, CA

Souders, Steve 2008 "High-Performance Web Sites"
Communications of the ACM 51, December 2008, p36-41.

WEB REFERENCES

[BC11] Adrian Ber and Alex Ciminian. *Patu Digua Open-Source Project*. **http://digua.sourceforge.net/**, 2011..

[Cim12] Alex Ciminian. *Usage Example: Hacking the CSS Grammar*. **https://gist.github.com/1738814**, 2012.

[Cro03] Douglas Crockford. *Jsmin*.
http://www.crockford.com/javascript/jsmin.html, 2003.

[Edw07] Dean Edwards. *Packer*.
http://dean.edwards.name/packer/, 2007.

[Goo10] Google. *Closure compiler*.
http://code.google.com/closure/compiler/, 2010.

[Ora11] Oracle. *Java servlet technology*.
http://www.oracle.com/technetwork/java/javaee/servlet/index.html, 2011.

[Scy04] Joe Scylla. *CSSmin*.
http://code.google.com/p/cssmin/, 2004.

[Too07] Dojo Toolkit. *Dojo shrinksafe*.
http://shrinksafe.dojotoolkit.org/, 2007.

[Wik11] Wikipedia. *Positional Notation*.
http://en.wikipedia.org/wiki/Positional_notation.

[Yah08] Yahoo! *YUI compressor*.
http://developer.yahoo.com/yui/compressor/, 2008