

# Improving Drupal's page loading performance

Thesis proposed to achieve the degree of bachelor  
in computer science/ICT/knowledge technology

Wim Leers

*Promotor:* Prof. dr. Wim Lamotte

*Co-promotor:* dr. Peter Quax

*Mentors:* Stijn Agten & Maarten Wijnants

Hasselt University  
Academic year 2008-2009

# Contents

<b>1</b>	<b>Abstract</b>	<b>4</b>
<b>2</b>	<b>Preface</b>	<b>4</b>
<b>3</b>	<b>Dutch summary/Nederlandstalige samenvatting</b>	<b>4</b>
<b>4</b>	<b>Terminology</b>	<b>5</b>
<b>5</b>	<b>Definition</b>	<b>6</b>
<b>6</b>	<b>Why it matters</b>	<b>7</b>
<b>7</b>	<b>The State of Drupal's page loading performance</b>	<b>8</b>
<b>8</b>	<b>Key Properties of a CDN</b>	<b>9</b>
<b>9</b>	<b>Profiling tools</b>	<b>10</b>
9.1	UA Profiler . . . . .	10
9.2	Cuzillion . . . . .	10
9.3	YSlow . . . . .	11
9.4	Hammerhead . . . . .	13
9.5	Apache JMeter . . . . .	15
9.6	Gomez/Keynote/WebMetrics/Pingdom . . . . .	16
9.6.1	Limited number of measurement points . . . . .	16
9.6.2	No real-world browsers . . . . .	16
9.6.3	Unsuited for Web 2.0 . . . . .	17
9.6.4	Paid & closed source . . . . .	17
9.7	Jiffy/Episodes . . . . .	17
9.7.1	Jiffy . . . . .	17
9.7.2	Episodes . . . . .	18
9.8	Conclusion . . . . .	21

<b>10 Improving Drupal: Episodes module</b>	<b>22</b>
10.1 The goal	22
10.2 Making episodes.js reusable	24
10.3 Episodes module: integrating with Drupal	24
10.3.1 Implementation	24
10.3.2 Screenshots	26
10.4 Episodes Server module: reports	29
10.4.1 Implementation	29
10.4.2 Screenshots	30
10.4.3 Desired future features	31
10.5 Insights	32
10.6 Feedback from Steve Souders	32
<b>11 Daemon</b>	<b>33</b>
11.1 Goals	33
11.2 Configuration file design	35
11.3 Python modules	36
11.3.1 filter.py	36
11.3.2 pathscanner.py	38
11.3.3 fsmonitor.py	38
11.3.4 persistent_queue.py and persistent_list.py	40
11.3.5 Processors	40
11.3.6 Transporters	45
11.3.7 config.py	48
11.3.8 daemon_thread_runner.py	49
11.4 Putting it all together: arbitrator.py	49

11.4.1	The big picture . . . . .	49
11.4.2	The flow . . . . .	50
11.4.3	Pipeline design pattern . . . . .	52
11.5	Performance tests . . . . .	55
11.6	Possible further optimizations . . . . .	55
11.7	Desired future features . . . . .	56
<b>12</b>	<b>Improving Drupal: CDN integration</b>	<b>57</b>
12.1	Goals . . . . .	57
12.2	Drupal core patch . . . . .	58
12.3	Implementation . . . . .	58
12.4	Comparison with the old CDN integration module . . . . .	59
12.5	Screenshots . . . . .	60
<b>13</b>	<b>Used technologies</b>	<b>66</b>
<b>14</b>	<b>Feedback from businesses</b>	<b>67</b>
<b>15</b>	<b>Conclusion</b>	<b>68</b>

## **1 Abstract**

TODO

## **2 Preface**

TODO

## **3 Dutch summary/Nederlandstalige samenvatting**

TODO

## 4 Terminology

**above the fold** The initially visible part of a web page: the part that you can see without scrolling

**AHAH** Asynchronous HTML And HTTP. Similar to AJAX, but the transferred content is HTML instead of XML.

**base path** The relative path in a URL that defined the root of a web site. E.g. if the site `http://yoursite.com/` is where a web site lives, then the base path is `/`. If you've got another web site at `http://yoursite.com/subsite/`, then the base path for that web site is `/subsite/`.

**browser** A web browser is an application that runs on end user computers to view web sites (which live on the World Wide Web). Examples are Firefox, Internet Explorer, Safari and Opera.

**CDN** A content delivery network (CDN) is a collection of web servers distributed across multiple locations to deliver content more efficiently to users. The server selected for delivering content to a specific user is typically based on a measure of network proximity.

**component** A component of a web page, this can be a CSS style sheet, a JavaScript file, an image, a font, a movie file, etc.

**document root** The absolute path on the file system of the web server that corresponds with the root directory of a web site. This is typically something like `/htdocs/yoursite.com`.

**episode** An episode in the page loading sequence.

**Episodes** The Episodes framework [40] (note the capital 'e').

**page loading performance** The time it takes to load a web page and all its components.

**page rendering performance** The time the server needs to render a web page.

**PoP** A Point of Presence is an access point to the internet; where multiple Internet Service Providers connect with each other.

**SLA** Service-Level Agreement, part of a service contract where the level of service is formally defined. In practice, the term SLA is sometimes used to refer to the contracted delivery time (of the service) or performance.

**web page** An (X)HTML document that potentially references components.

## 5 Definition

When an end user loads a web page, the time perceived by him until the page has loaded entirely is called the *end user response time*. Unlike what you might think, the majority of this time isn't spent at the server, generating the page! The generating (back-end) and transport of the HTML document (front-end) is typically only 10-20% of the end user response time. The other 80-90% of the time is spent on loading the components (CSS stylesheets, JavaScript, images, movies, etc.) in the page (front-end only). Figure 1 clarifies this visually:

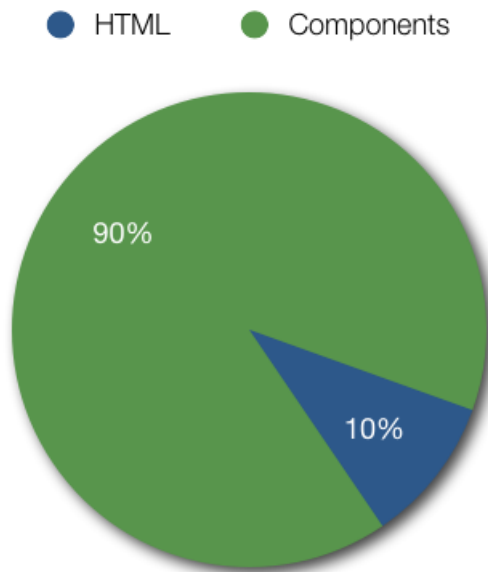


Figure 1: End user response time of a typical web page.

It should be obvious now that it's far more effective to focus on front-end performance than it is to focus on back-end performance, because it's got a greater potential. It's also easier to optimize than the back-end.

## 6 Why it matters

Page loading performance matters for a single reason:

Users care about performance!

Your web site's visitors won't be timing the page loads themselves, but they will browse elsewhere when you're forcing them to wait too long. Fast web sites are rewarded, slow web sites are punished. Fast web sites get more visitors, have happier visitors and their visitors return more often. If the revenue of your company is generated through your web site, you'll want to make sure that page loading performance is as good as possible, because it will maximize your revenue as well.

Some statistics:

- Amazon: 100 ms of extra load time caused a 1% drop in sales [1]
- Yahoo!: 400 ms of extra load time caused a 5-9% drop in full-page traffic (meaning that they leave before the page has finished loading) [1]
- Google: 500 ms of extra load time caused 20% fewer searches [1]
- Google: trimming page size by 30% resulted in 30% more map requests [2]

It's clear that even the smallest delays can have disastrous and wondrous effects.

Now, why is this important to Drupal – because this bachelor thesis is about improving Drupal's page loading performance in particular? Because then the Drupal Experience is better: a faster web site results in happier users and developers. If your site is a commercial one – either through ads or a store, then it also impacts your revenue. More generally, a faster Drupal would affect many:

- Drupal is increasingly used for big, high-traffic web sites, thus a faster Drupal would affect a lot of people
- Drupal is still growing in popularity (according to its usage statistics, which only includes web sites with the Update Status module enabled, there are over 140,000 web sites as of February 22, 2009, see [4]) and would therefore affect ever more people
- Drupal is international, thanks to its i18n/L10n support, and thanks to that it's used for sites with very international audiences (whom face high network latencies) and developing countries (where low-speed internet connections are commonplace). A faster Drupal would make a big difference there as well.



## 7 The State of Drupal's page loading performance

So you might expect that Drupal has already invested heavily in improving its page loading performance. Unfortunately, that's not true. Hopefully this bachelor thesis will help to gain some developer attention.

Because of this, the article I wrote more than a year ago is still completely applicable. It doesn't make much sense to just rephrase the article here in my thesis text, so instead I'd like to forward you to that article [5].

I've repeated the remaining problems here for the sake of clarity:

- Static files (CSS, JavaScript, images) should be served with proper HTTP headers so that the browser can cache them and reduce the number of HTTP requests for each page load. Especially the Expires header is important here.
- To allow for CDN integration in Drupal, the ability to alter file URLs dynamically is needed, but this isn't supported yet.
- CSS and JS files should be served GZIPped when the browser supports it.
- JavaScript files should be at the bottom (just before the closing `</body>` tag) whenever possible.
- JavaScript files should be minified.
- Drupal should provide a mechanism to render the same content in multiple formats: (X)HTML (for the regular browser), partial HTML or JSON (for AHAH), XML (for AJAX) and so on. You should be able to set transformations, including cacheability and GZIPability per format.
- CSS sprites should be generated automatically.

## 8 Key Properties of a CDN

I'll repeat the definition from the terminology section:

A content delivery network (CDN) is a collection of web servers distributed across multiple locations to deliver content more efficiently to users. The server selected for delivering content to a specific user is typically based on a measure of network proximity.

It's extremely hard to decide which CDN to use. In fact, by just looking at a CDN's performance, it's close to impossible [6, 7]!

So you must look at the featureset – instead of performance – when deciding. Depending on your audience, the geographical spread (the number of PoPs around the world) may be very important to you. A 100% SLA is also nice to have.

You may also choose a CDN based on the population methods it supports. There are two big categories here: *push* and *pull*. Pull requires virtually no work on your side: all you have to do, is rewrite the URLs to your files. The CDN will then apply the *Origin Pull* technique and will periodically pull the files from the origin (that's your server). How often that is, depends on how you've configured headers (particularly the Expires headers). It of course also depends on the software driving the CDN – there is no standard in this field. It may also result in redundant traffic because files are being pulled from the origin server more often than they actually change, but this is a minor drawback in most situations. Push on the other hand, requires a fair amount of work on your side to sync files to the CDN. But you gain flexibility because you can decide when files are synced, how often and if any preprocessing should happen. That's much harder to do with Origin Pull CDNs. See table 1 for an overview on this.

It should also be noted that some CDNs, if not most, support both Origin Pull and one or more push methods.

The last thing to consider is vendor lock-in. Some CDNs offer highly specialized features, such as video transcoding. If you then discover another CDN that is significantly cheaper, you cannot easily move, because you're depending on your current CDN's specific features.

	PULL	PUSH
TRANSFER PROTOCOL	none	FTP, SFTP, WebDAV, Amazon S3 ...
ADVANTAGES	virtually no setup	flexibility, no redundant traffic
DISADVANTAGES	no flexibility, redundant traffic	setup

Table 1: Pull versus Push CDNs comparison chart.

## 9 Profiling tools

*If you can not measure it, you can not improve it.*

Lord Kelvin

The same applies to page loading performance: if you can't measure it, you can't know which parts have the biggest effect and thus deserve your focus. So before doing any real work, we'll have to figure out which tools can help us analyzing page loading performance. "profiling" turns out to be a more accurate description than "analyzing":

In software engineering, *performance analysis*, more commonly today known as *profiling*, is the investigation of a program's behavior using information gathered as the program executes. The *usual goal of performance analysis is to determine which sections of a program to optimize* — usually either to increase its speed or decrease its memory requirement (or sometimes both).

So we'll go through a list of tools: UA Profiler, Cuzillion, YSlow, Hammerhead, Apache JMeter, Gomez/Keynote/WebMetrics/Pingdom and Jiffy/Episodes. As you can tell, it's a pretty long list, so I'll pick the tools I'll use while improving Drupal's page loading performance based on two factors:

1. How the tool could help improve Drupal core's page loading performance.
2. How the tool could help Drupal site owners profile their site's page loading performance.

### 9.1 UA Profiler

UA Profiler [8] is a crowd-sourced project for gathering browser performance characteristics (parallel connections, downloading scripts without blocking, caching, etc.). The tests run automatically when you navigate to the test page from any browser — this is why it's powered by crowdsourcing.

It's a handy, informative tool to find out which browser supports which features related to page loading performance.

### 9.2 Cuzillion

Cuzillion [9] was introduced [10] on April 25, 2008 so it is a relatively new tool. Its tag line, "*cuz there are zillion pages to check*" indicates what it's about: there are a lot of possible combinations of stylesheets, scripts and images. Plus they can be external or inline. And each combination has different effects. Finally, to further complicate the situation, all these combinations depend on the browser being used. It should be obvious that without Cuzillion, it's an insane job to figure out how each browser behaves:

Before I would open an editor and build some test pages. Firing up a packet sniffer I would load these pages in different browsers to diagnose what was going on. I was starting my research on advanced techniques for loading scripts without blocking and realized the number of test pages needed to cover all the permutations was in the hundreds. That was the birth of Cuzillion.

Cuzillion is not a tool that helps you analyze any *existing* web page. Instead, it allows you to analyze any combination of components. That means it's a *learning tool*. You could also look at it as a *browser profiling tool* instead of all other listed tools, which are *page loading profiling tools*.

Let's go through a simple example for a better understanding. How does the following combination of components (in the `<body>` tag) behave in different browsers?

1. an image on domain 1 with a 2 second delay
2. an inline script with a 2 second execution time
3. an image on domain 1 with a 2 second delay

First you create this setup in Cuzillion (see [figure 2 on the following page](#)). This generates a unique URL. You can then copy this URL to all browsers you'd like to test.

As you can see, Safari and Firefox behave very differently. In Safari (see [figure 3 on the next page](#)), the loading of the first image seems to be deferred until the inline script has been executed. In Firefox (see [figure 4 on page 13](#)), the first image is immediately rendered and after a delay of 2 seconds – indeed the execution time of the inline script – the second image is rendered. Without going into details about this, it should be clear that Cuzillion is a simple, yet powerful tool to learn about browser behavior, which can in turn help to improve the page loading performance.

### 9.3 YSlow

YSlow [15] also is a Firebug [13] extension (see [5 on page 14](#)) that can be used to analyze page loading performance through 13 rules. These were part of the original 14 rules [17] – of which there are now 34 – of “Exceptional Performance” [16], as developed the Yahoo! performance team.

YSlow 1.0 can only evaluate these 13 rules and has a hardcoded grading algorithm. You should also remember that YSlow just checks how well a web page implements these rules. It analyzes the content of your web page (and the headers that were sent with it). For example, it doesn't test the latency or speed of a CDN, it just checks if you are using one. As an example, because you have

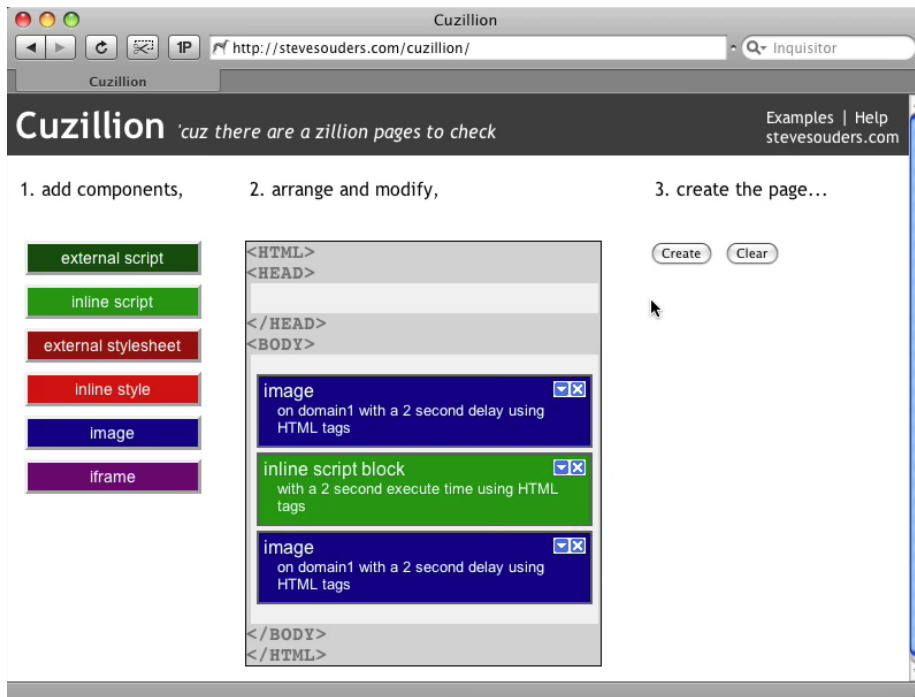


Figure 2: The example situation created in Cuzillion.

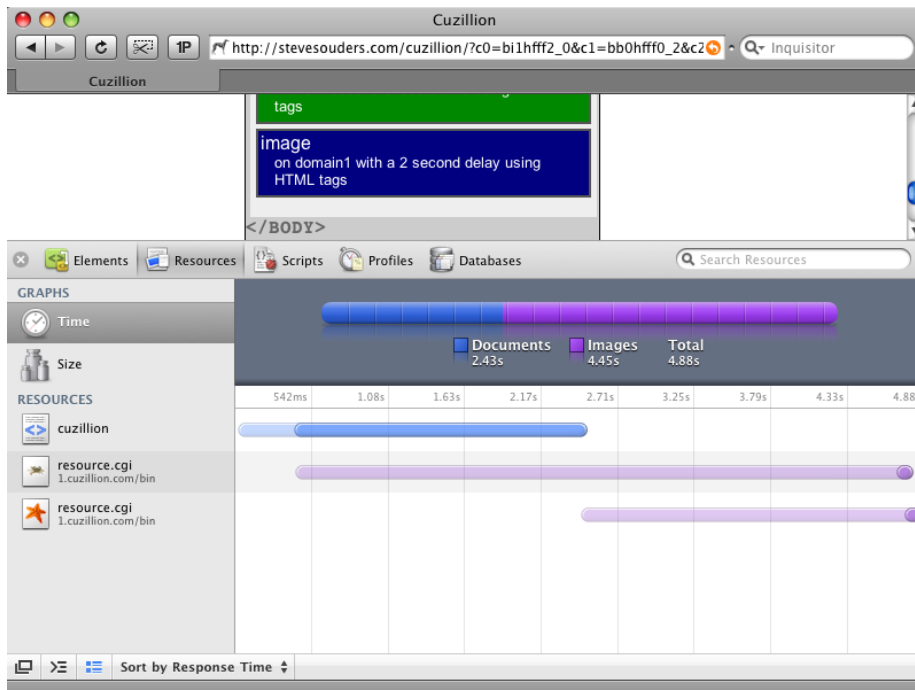


Figure 3: The example situation in Safari 3.

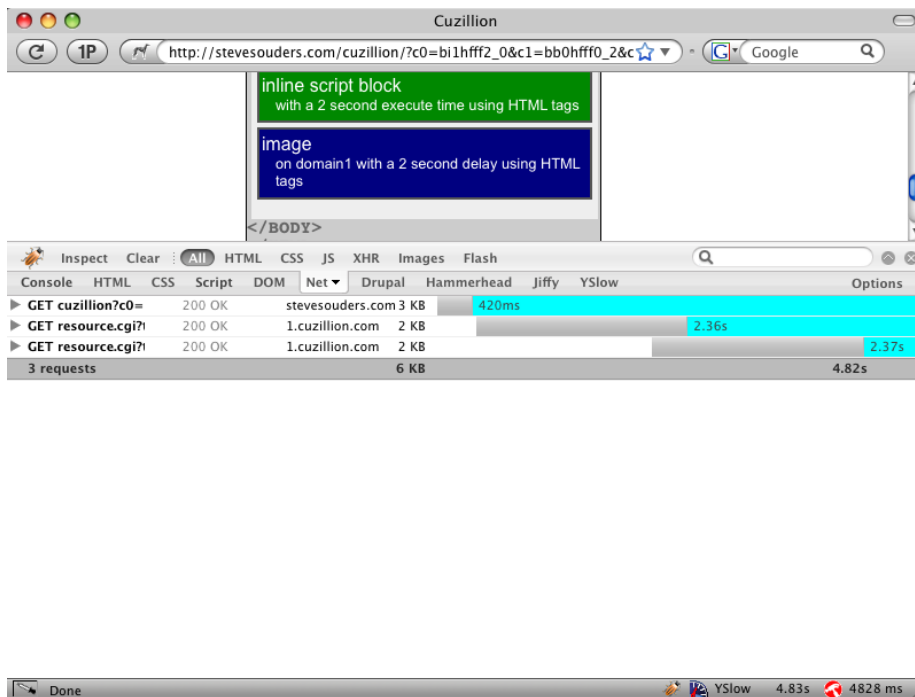


Figure 4: The example situation in Firefox 3.

to tell YSlow (via Firefox' `about:config`) what the domain name of your CDN is, you can even fool YSlow into thinking any site is using a CDN: see [6 on the following page](#).

That, and the fact that some of the rules it analyzes are only relevant to *very* big web site. For example, one of the rules (#13, "Configure ETags") is only relevant if you're using a cluster of web servers. For a more in-depth article on how to deal with YSlow's evaluation of your web sites, see [18]. YSlow 2.0 [19] aims to be more extensible and customizable: it will allow for community contributions, or even web site specific rules.

Since only YSlow 1.0 is available at the time of writing, I'll stick with that. It's a very powerful and helpful tool as it stands, it'll just get better. But remember the two caveats: it only verifies rules (it doesn't measure real-world performance) and some of the rules may not be relevant for your web site.

## 9.4 Hammerhead

Hammerhead [11, 12] is a Firebug [13] extension that should be used *while* developing. It measures how long a page takes to load and it can load a page multiple times, to calculate the average and mean page load times. Of course, this is a lot less precise than real-world profiling, but it allows you to profile while you're working. It's far more effective to prevent page loading performance problems

Performance Grade: F (57)	
F	1. Make fewer HTTP requests <span>▾</span> <b>This page has 4 external JavaScript files.</b> <b>This page has 31 CSS background images.</b>
F	2. Use a CDN <span>▾</span>
F	3. Add an Expires header <span>▾</span>
A	4. Gzip components
A	5. Put CSS at the top
B	6. Put JS at the bottom <span>▾</span> <b>2 external scripts were found in the document HEAD. Could they be moved lower in the page?</b> <a href="http://drupal.org/misc/jquery.js">http://drupal.org/misc/jquery.js</a> <a href="http://drupal.org/misc/drupal.js">http://drupal.org/misc/drupal.js</a>
A	7. Avoid CSS expressions
n/a	8. Make JS and CSS external <span>▾</span>
A	9. Reduce DNS lookups
B	10. Minify JS <span>▾</span>
A	11. Avoid redirects
A	12. Remove duplicate scripts
F	13. Configure ETags <span>▾</span>

Figure 5: YSlow applied to drupal.org.

Performance Grade: F (59)		Performance Grade: C (70)	
F	1. Make fewer HTTP requests <span>▾</span>	F	1. Make fewer HTTP requests <span>▾</span>
<b>F</b>	2. Use a CDN <span>▾</span>	<b>A</b>	2. Use a CDN <span>▾</span>
F	3. Add an Expires header <span>▾</span>	F	3. Add an Expires header <span>▾</span>
A	4. Gzip components	A	4. Gzip components
A	5. Put CSS at the top	A	5. Put CSS at the top
A	6. Put JS at the bottom	A	6. Put JS at the bottom
A	7. Avoid CSS expressions	A	7. Avoid CSS expressions
n/a	8. Make JS and CSS external <span>▾</span>	n/a	8. Make JS and CSS external <span>▾</span>
A	9. Reduce DNS lookups	A	9. Reduce DNS lookups
A	10. Minify JS <span>▾</span>	A	10. Minify JS <span>▾</span>
A	11. Avoid redirects	A	11. Avoid redirects
A	12. Remove duplicate scripts	A	12. Remove duplicate scripts
F	13. Configure ETags <span>▾</span>	F	13. Configure ETags <span>▾</span>

(a) The original YSlow analysis.      (b) The resulting YSlow analysis.

Figure 6: Tricking YSlow into thinking drupal.org is using a CDN.

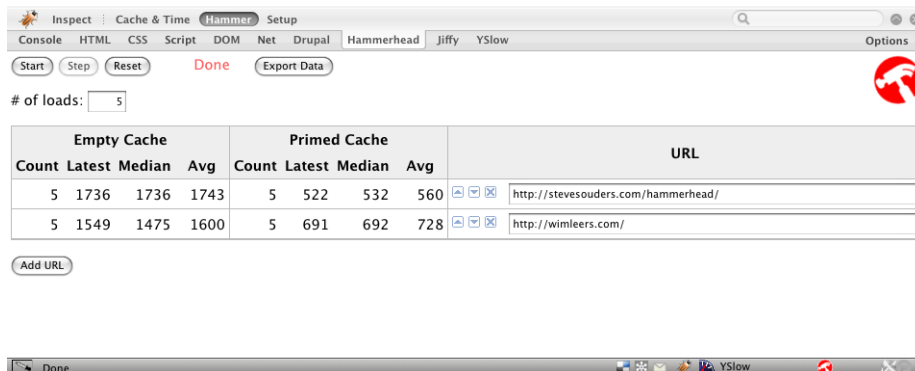


Figure 7: Hammerhead.

due to changes in code, because you have the test results within seconds or minutes after you've made these changes!

Of course, you could also use YSlow (see section 9.3) or FasterFox [14], but then you have to load the page multiple times (i.e. *hammer* the server, this is where the name comes from). And you'd still have to set up the separate testing conditions for each page load: empty cache, primed cache and for the latter there are again two possible situations: disk cache *and* memory cache or just disk cache. Memory cache is of course faster than disk cache; that's also why that distinction is important. Finally, it supports exporting the resulting data into CSV format, so you could even create some tools to roughly track page loading performance throughout time. See figure 7.

## 9.5 Apache JMeter

Apache JMeter [21] is an application *designed to load test functional behavior and measure performance*. In the perspective of profiling page loading performance, the relevant features are: loading of web pages with and without its components and measuring the response time of just the HTML or the HTML and all the components it references.

However, it has several severe limitations:

- Because it only measures from one location – the location from where it's run, it doesn't give a good big picture.
- It isn't an actual browser, so it doesn't download components referenced from CSS or JS files.
- Also because it isn't an actual browser, it doesn't behave the same as browsers when it comes to parallel downloads.



- It requires more setup than Hammerhead (see [9.4 on page 13](#)), so it's less likely that a developer will make JMeter part of his workflow.

It can be very useful in case you're doing performance testing (How long does the back-end need to generate certain pages?), load testing (How many concurrent users can the back-end/server setup handle?) and stress testing (How many concurrent users can it handle until errors ensue?)

To learn more about load testing Drupal with Apache JMeter, see [\[22, 23\]](#)

## 9.6 Gomez/Keynote/WebMetrics/Pingdom

Gomez [\[24\]](#), KeyNote [\[25\]](#), WebMetrics [\[26\]](#) and Pingdom [\[27\]](#) are examples of 3rd-party (paid) performance monitoring systems.

They have four major disadvantages:

1. limited number of measurement points
2. no real-world browsers are used
3. unsuited for Web 2.0
4. paid & closed source

### 9.6.1 Limited number of measurement points

These services poll your site at regular or irregular intervals. This poses analysis problems: for example, if one your servers is very slow just at that one moment that any of these services requests a page, you will be told that there's a major issue with your site. But that's not necessarily true: it might be a fluke.

### 9.6.2 No real-world browsers

Most, if not all of these services use their own custom clients [\[34\]](#). That implies their results aren't a representation of the real-world situation, which means you can't rely upon these metrics for making decisions: what if a commonly used real-world browser behaves completely differently? Even if the services would all use real-world browsers, they would never reflect real-world performance, because browser usage evolves over time and aren't the same for any two web sites.

### 9.6.3 Unsited for Web 2.0

The problem with these services is that they still assume the World Wide Web is the same as it was 10 years ago, where JavaScript was rather a scarcity than the abundance it is today. They still interpret the `onload` event as the “end time” for response time measurements. In Web 1.0, that was fine. But as the adoption of AJAX [28] has grown, the `onload` event has become less and less representative of when the page is ready (i.e. has completely loaded), because the page can continue to load additional components. For some web sites, the “above the fold” section of a web page has been optimized, thereby loading “heavier” content later, below the fold. Thus the “page ready” point in time is shifted from its default.

In both of these cases, the `onload` event is too optimistic [37].

There are two ways to measure Web 2.0 web sites [38]:

1. *manual scripting*: identify timing points using scripting tools (Selenium [29], Keynote’s KITE [30], etc.). This approach has a long list of disadvantages: low accuracy, high switching costs, high maintenance costs, synthetic (no real-world measurements).
2. *programmable scripting*: timing points are marked by JavaScript (Jiffy [35], Gomez Script Recorder [31], etc.). This is the preferred approach: it has lower maintenance costs and a higher accuracy because the code for timing is included in the other code and measures real user traffic. If we would now work on a shared implementation of this approach, then we wouldn’t have to reinvent the wheel every time and switching costs would be much lower. See the Jiffy/Episodes later on.

### 9.6.4 Paid & closed source

You’re dependent upon the 3rd party service to implement new instrumentations and analyses. There’s a high cost for the implementation cost and a very high cost when switching to a different 3rd party service.

## 9.7 Jiffy/Episodes

### 9.7.1 Jiffy

Jiffy [33, 34, 35] is designed to give you real-world information on what’s actually happening within browsers of users that are visiting your site. It shows you how long pages really take to load and how long events that happen while or after your page is loading really take. Especially when you don’t control all the components of your web site (e.g. widgets of photo and music web

sites, contextual ads or web analytics services), it's important that you can monitor their performance. It overcomes to 4 major disadvantages that were listed previously:

1. it can measure *every* page load if desired
2. real-world browsers are used, because it's just JavaScript code that runs in the browser
3. well-suited for Web 2.0, because you can configure it to measure *anything*
4. open source

Jiffy consists of several components:

- **Jiffy.js**: a library for instrumenting your pages and reporting measurements
- Apache configuration: to receive and log measurements
- Ingestor: parse logs and store in a database (currently only supports Oracle XE)
- Reporting toolset
- Firebug extension [36], see figure 8 on the next page

Jiffy was built to be used by the WhitePages web site [32] and has been running on that site. At more than 10 million page views per day, it should be clear that Jiffy can scale quite well. It's been released as an open source project, but at the time of writing, the last commit was on July 25, 2008. So it's a dead project.

### 9.7.2 Episodes

Episodes [40, 41] is very much like Jiffy. There are two differences:

1. Episodes' goal is to become an industry standard. This would imply that the aforementioned 3rd party services (Gomez/Keynote/WebMetrics/Pingdom) would take advantage of the the instrumentations implemented through Episodes in their analyses.
2. Most of the implementation is built into browsers (`window.postMessage()`, `addEventListener()`), which means there is less code that must be downloaded. (Note: the newest versions of browsers are necessary: Internet Explorer 8, Firefox 3, WebKit Nightlies and Opera 9.5. An additional backwards compatibility JavaScript file must be downloaded for older browsers.)

The screenshot shows a web browser window titled "Verify Jiffy Extension - Carousel Example" with the URL "http://billwscott.com/jiffyext/testJiffyExt.html". The page content includes a heading "Verify Jiffy Extension Installation" and a paragraph explaining how to use the extension. Below this is a carousel of four travel-related images with captions: "Ghent, Prague and Amsterdam: I can't believe we were in transit a...", "Our trip to usa This is just the bas...", "Europe Backpacking Trip 2008 So, I left Boston la...", and "Thanksgiving in NYC 2008 We left our hotel in...".

At the bottom of the browser window, the Jiffy extension's performance monitoring interface is visible. It includes a search bar and a list of performance events with their durations:

Event	Duration
PageStart	2071 ms
DOMContentLoaded	2060 ms
load	6 ms
Finished onLoad	5 ms
Fetch	887 ms
AjaxRequestMade	1 ms
ResponseReceived	886 ms
Load Items	1 ms
EndLoad	1 ms
Carousel	4 ms
Finished Creating	4 ms

Figure 8: Jiffy.

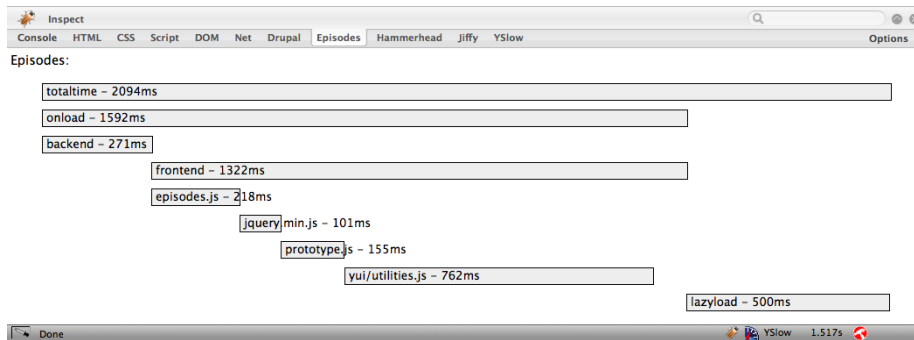


Figure 9: Episodes.

Steve Souders outlines the goals and vision for Episodes succinctly in these two paragraphs:

*The goal is to make Episodes the industrywide solution for measuring web page load times.* This is possible because Episodes has benefits for all the stakeholders. Web developers only need to learn and deploy a single framework. Tool developers and web metrics service providers get more accurate timing information by relying on instrumentation inserted by the developer of the web page. Browser developers gain insight into what's happening in the web page by relying on the context relayed by Episodes.

Most importantly, users benefit by the adoption of Episodes. They get a browser that can better inform them of the web page's status for Web 2.0 apps. Since Episodes is a lighter weight design than other instrumentation frameworks, users get faster pages. As Episodes makes it easier for web developers to shine a light on performance issues, *the end result is an Internet experience that is faster for everyone.*

A couple of things can be said about the current codebase of Episodes:

- There are two JavaScript files: `episodes.js` and `episodes-compat.js`. The latter is loaded on-the-fly when an older browser is being used that doesn't support `window.postMessage()`. These are operational but haven't had wide testing yet.
- It uses the same query string syntax as Jiffy uses, which means Jiffy's Apache configuration, ingestor and reporting toolset can be reused, at least partially.
- It has its own Firebug extension, see figure 9.

So, Episodes' very *raison d'être* is to achieve a consensus on a JavaScript-based page loading instrumentation toolset. It aims to become an industry

standard and is maintained by Steve Souders, who is currently on Google's payroll to work on all things page loading performance full-time (which suggests we might see integration with Google's Analytics [39] service in the long term). Add in the fact that Jiffy hasn't been updated since its initial release, and it becomes clear that Episodes is the better long-term choice.

## 9.8 Conclusion

There isn't a single, "do-it-all" tool that you should use. Instead, you should wisely combine all of the above tools. Use the tool that fits the task at hand.

However, for the scope of this thesis, there is one tool that jumps out: YSlow. It allows you to carefully analyze which things Drupal could be doing better. It's not necessarily meaningful in real-world situations, because it e.g. only checks if you're using a CDN, not how fast that CDN is. However, the fact that it tests whether a CDN is being used (or Expired headers, or gzipped components, or ...) is enough to find out what can be improved, to maximize the potential performance.

This kind of analysis is exactly what I'll perform in the next section: [7 on page 8](#).

There is one more tool that jumps out for real, practical use: Episodes. This tool, if properly integrated with Drupal, would be a key asset to Drupal, because it would enable web site owners to track the real-world page loading performance. It would allow contributed module developers to support Episodes. This, in turn, would be a good indicator for a module's quality and would allow the web site owner/administrator/developer to carefully analyze each aspect of his Drupal web site.

I will create this integration as part of my bachelor thesis, see section [10 on the following page](#).

## 10 Improving Drupal: Episodes module

The work I'll be doing on improving Drupal's page loading performance should be practical, not theoretical. It should have a real-world impact.

To ensure that that also happens, I wrote the Episodes module [42]. This module integrates the Episodes framework for timing web pages (see section 9.7.2 on page 18) with Drupal on several levels – *all without modifying Drupal core*:

- Automatically includes the necessary JavaScript files and settings on each appropriate page.
- Automatically inserts the crucial initialization variables at the beginning of the `head` tag.
- Automatically turns each behavior (in `Drupal.behaviors`) is automatically into its own episode.
- Provides a centralized mechanism for lazy loading callbacks that perform the lazy loading of content. These are then also automatically measured.
- For measuring the `css`, `headerjs` and `footerjs` episodes, you need to change a couple of lines in the `page.tpl.php` file of your theme.
- Provides basic reports with charts to make sense of the collected data.

The Episodes module in fact contains two modules: the Episodes module and the Episodes Server module. The former is the *actual* integration and can be used without the latter. The latter can be installed on a separate Drupal web site (on a separate server) or on the same. It provides basic reports.

You could also choose to not enable the Episodes Server module and use an external web service to generate reports, but for now, none of these services exist yet. That's a void that will probably be filled in the next few years by the business world.

### 10.1 The goal

The goal is to measure the different episodes of loading a web page. Let me clarify that via a timeline, while referencing the HTML in listing 1 on the following page.

The main measurement points are:

- `starttime`: time of requesting the web page (when the `unbeforeunload` event fires, the time is stored in a cookie); not in the HTML file

Listing 1: Sample Drupal HTML file.

```

1 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
2   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
3 <html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en" dir="ltr">
4   <head>
5     <title>Sample Drupal HTML</title>
6     <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
7     <link rel="shortcut icon" href="/misc/favicon.ico" type="image/x-icon" />
8     <link type="text/css" rel="stylesheet" media="all" href="main.css" />
9     <link type="text/css" rel="stylesheet" media="print" href="more.css" />
10    <script type="text/javascript" src="main.js"></script>
11    <script type="text/javascript">
12      <!--//--><![CDATA[//><!--
13      jQuery.extend(Drupal.settings, { "basePath": "/drupal/", "more": true });
14      //--><![]]>
15    </script>
16    <!--[if lt IE 7]>
17      <link type="text/css" rel="stylesheet" media="all" href="fix-ie.css" />
18    <![endif]-->
19  </head>
20  <body>
21    <!--
22      lots
23      of
24      HTML
25      here
26    -->
27    <script type="text/javascript" src="more.js"></script>
28  </body>
29 </html>

```

- firstbyte: time of arrival of the first byte of the HTML file (the JavaScript to measure this time should be as early in the HTML as possible for highest possible accuracy); line 1 of the HTML file
- domready: when the entire HTML document is loaded, but *just* the HTML, *not* the referenced files
- pageready: when the `onload` event fires, this happens when also all referenced files are loaded
- totaltime: when everything, also the possible lazy-loaded content, is loaded (i.e. pageready + the time to lazy-load content)

Which make for these basic episodes:

- backend episode = firstbyte - starttime
- frontend episode = pageready - firstbyte
- domready episode = domready - firstbyte, this episode is contained within the frontend episode
- totaltime episode = totaltime - starttime, this episode contains the backend and frontend episodes

These are just the basic time measurements and episodes. It's possible to also measure the time it took to load the CSS (lines 8-9) and JS files in the header



```

1 <head>
2
3 <!-- Initialize EPISODES. -->
4 <script type="text/javascript">
5     var EPISODES = EPISODES || {};
6     EPISODES.frontendStartTime = Number(new Date());
7     EPISODES.compatScriptUrl = "lib/episodes-compat.js";
8     EPISODES.logging = true;
9     EPISODES.beaconUrl = "episodes/beacon";
10 </script>
11
12 <!-- Load episodes.js. -->
13 <script type="text/javascript" src="lib/episodes.js" />
14
15 <!-- Rest of head tag. -->
16 <!-- ... -->
17
18 </head>

```

(line 10) and in the footer (line 27), for example. It's possible to measure just about anything you want.

For a visual example of all the above, see [figure 12 on page 28](#).

## 10.2 Making episodes.js reusable

The `episodes.js` file provided at the Episodes example [43] was in fact just a rough sample implementation, an implementation that indicates what it should look like. It contained several hardcoded URLs, didn't measure the sensible default episodes, contained a few bugs. In short, it was an excellent and solid start, but it needed some work to be truly reusable.

I improved `episodes.js` to make it reusable, so that I could integrate it with Drupal without adding Drupal-specific code to it. I made it so that all you have to do is something like this:

This way, you can set the variables to what you need them to be without customizing `episodes.js`. Line 6 should be as early in the page as possible, because it's the most important reference timestamp.

## 10.3 Episodes module: integrating with Drupal

### 10.3.1 Implementation

Here's a brief overview with the highlights of what had to be done to integrate Drupal with the Episodes framework.

- Implemented `hook_install()`, through which I set a module weight of -1000. This ensures the hook implementations of this module are always executed before all others.

Listing 2: `Drupal.attachBehaviors()` override.

```
Drupal.attachBehaviors = function(context) {
  url = document.location;

  for (behavior in Drupal.behaviors) {
    window.postMessage("EPISODES:mark:" + behavior, url);
    Drupal.behaviors[behavior](context);
    window.postMessage("EPISODES:measure:" + behavior, url);
  }
};
```

- Implemented `hook_init()`, which is invoked at the end of the Drupal bootstrap process. Through this hook I automatically insert the JavaScript into the `<head>` tag that's necessary to make Episodes work (see section [10.2 on the preceding page](#)). Thanks to the extremely low module weight, the JavaScript code it inserts is the first tag in the `<head>` tag.
- Also through this same hook I add `Drupal.episodes.js`, which provides the actual integration with Drupal. It automatically creates an episode for each Drupal "behavior". (A behavior is written in JavaScript and adds interactivity to the web page.) Each time new content is added to the page through AHAH, `Drupal.attachBehaviors()` is called and automatically attaches behaviors to new content, but not to existing content. Through `Drupal.episodes.js`, Drupal's default `Drupal.attachBehaviors()` method is overridden – this is very easy in JavaScript. In this overridden version, each behavior is automatically measured as an episode. Thanks to Drupal's existing abstraction and the override I've implemented, all JavaScript code can be measured through Episodes without hacking Drupal core. A simplified version of what it does can be seen in listing [2](#).
- Some of the Drupal behaviors are too meaningless to measure, so it'd be nice to be able to mark some of the behaviors as ignored. That's also something I implemented. Basically I do this by locating every directory in which one or more `*.js` files exist, create a scan job for each of these and queue them in Drupal's Batch API [\[44\]](#). Each of these jobs scans each `*.js` file, looking for behaviors. Every detected behavior is stored in the database and can be marked as ignored through a simple UI that uses the Hierarchical Select module [\[46\]](#).
- For measuring the `css` and `headerjs` episodes, it's necessary to make a couple of simple (copy-and-paste simple) changes to the `page.tpl.php` of the Drupal theme(s) you're using. These changes are explained in the `README.txt` file that ships with the Episodes module. This is the only manual change to code that *can* be done – it's recommended, but not required.
- And of course a configuration UI (see figure [10 on the following page](#) and figure [11 on page 27](#)) using the Forms API [\[45\]](#). It ensures the beacon URL exists and is properly configured (i.e. returns a zero-byte file).

### 10.3.2 Screenshots

The screenshot shows the 'Episodes' settings page in a Drupal administration interface. The breadcrumb trail is 'Home > Administer > Site configuration'. The page has three tabs: 'Episodes', 'Settings' (which is active), and 'Behaviors'. Under the 'Settings' tab, there are three main sections: 'Status', 'Beacon URL', and 'Advanced settings'. The 'Status' section has three radio buttons: 'Disabled', 'Debug mode', and 'Enabled' (which is selected). Below this is a paragraph explaining that enabling Episodes applies to users with the 'administer site configuration' permission and that logging will be disabled. The 'Beacon URL' section has a text input field containing 'http://dev/d6/episodes/beacon' and a paragraph explaining that this is the URL used for logging analysis. The 'Advanced settings' section is expanded and contains two sub-sections: 'Logging' and 'Excluded paths'. The 'Logging' section has two radio buttons: 'Disabled' and 'Enabled' (which is selected), with a note that a valid beacon URL is required when logging is enabled. The 'Excluded paths' section has a text area containing 'admin/\*' and a paragraph explaining that paths are entered one per line, with '\*' as a wildcard and examples like 'blog' and 'blog/\*'. At the bottom of the form are two buttons: 'Save configuration' and 'Reset to defaults'.

Home > Administer > Site configuration

**Episodes** Settings Behaviors

**Status:**

Disabled

Debug mode

Enabled

You can either disable or enable Episodes, or put it in debug mode, in which case it will only be applied for users with the *administer site configuration* permission and logging will be disabled.

**Beacon URL:**

The beacon URL you would like to use. Can also be the URL of a different server, potentially a web service that analyzes the logs for you.

▼ **Advanced settings**

**Logging:**

Disabled

Enabled

When you've enabled logging, you should provide a valid beacon URL.

**Excluded paths:**

Enter one page per line as Drupal paths. The '\*' character is a wildcard. Example paths are *blog* for the blog page and *blog/\** for every personal blog. <front> is the front page.

Save configuration Reset to defaults

Figure 10: Episodes module settings form.

Home > Administer > Site configuration > Episodes

**Episodes**   Settings   **Behaviors**

- Scanned 178 files and found 35 behaviors.
- Updated 0 behaviors, detected 35 new behaviors.

**Detected behaviors**

**35 behaviors** in 12 modules, spread over 26 files were detected. If you've installed new modules or themes, you may want to scan again. *Scanning again will not reset the ignored behaviors!*  
 Last scanned: **0 sec ago.**

**Ignored behaviors:**

All ignored behaviors

collapse (core)	<a href="#">Remove</a>
multiselectSelector (core)	<a href="#">Remove</a>
viewsDependent (views)	<a href="#">Remove</a>

Select the behaviors that you **don't** want to be measured!

Figure 11: Episodes module behaviors settings form.

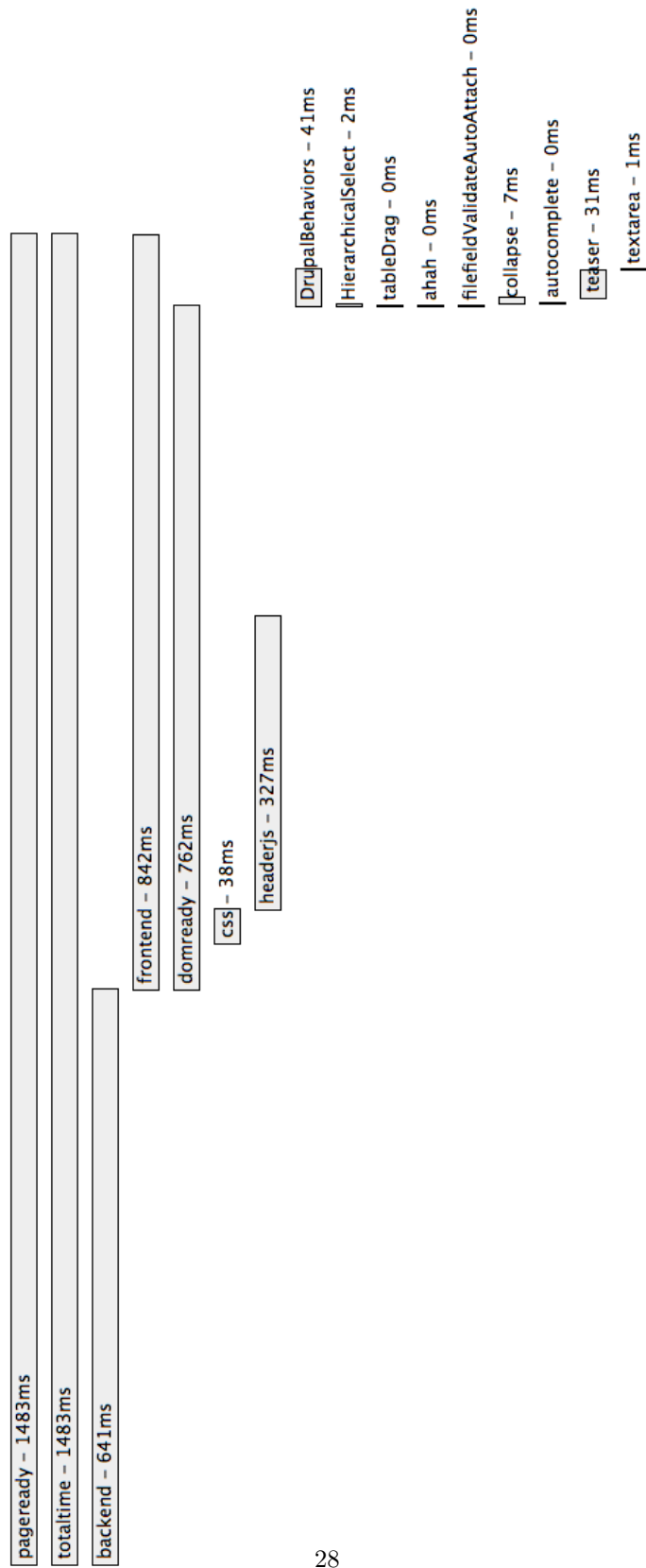


Figure 12: Results of Episodes module in the Episodes Firebug add-on.

## 10.4 Episodes Server module: reports

Only basic reports are provided, highlighting the most important statistics and visualizing them through charts. Advanced/detailed reports are beyond the scope of this bachelor thesis, because they require extensive performance research (to be able to handle massive datasets), database indexing optimization and usability research.

### 10.4.1 Implementation

- First of all, the Apache HTTP server is a requirement, this application's logging component is used for generating the log files. Its logging component has been proven to be scalable, so there's no need to roll our own. The source of this idea lies with Jiffy (see section [9.7.1 on page 17](#)).
- The user must make some changes to his `httpd.conf` configuration file for his Apache HTTP server. As just mentioned, my implementation is derived from Jiffy's, yet every configuration line is different.
- The ingestor parses the Apache log file and moves the data to the database. I was able to borrow a couple of regular expressions from Jiffy's ingestor (which is written in Perl ...) but I rewrote it completely to be clean and simple code, conform the Drupal coding guidelines. It detects the browser, browser version and operating system from the User Agent that was logged with the help of the Browser.php library [\[48\]](#). This is guaranteed to work thanks to the included meticulous unit tests.
- For the reports, I used the Google Chart API [\[47\]](#). You can see the result in [figure 14 on page 31](#).
- And of course again a configuration UI (see [figure 13 on the following page](#)) using the Forms API [\[45\]](#). It ensures the log file exists and is accessible for reading.

## 10.4.2 Screenshots

Home > Administer > Site configuration

### Episodes Server

**Ingestor settings**

Only when you are using a beacon URL that points to this same server and have set up Apache properly to do the logging, only then, you have to configure the ingestor. The ingestor runs during cron.

**Status:**

Disabled

Enabled

**Log file:**

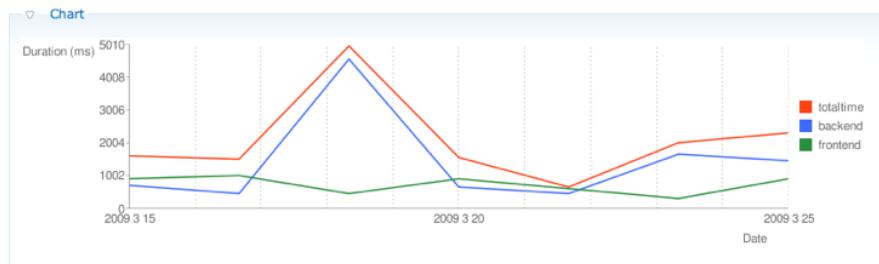
The **absolute path** to the log file that must be Ingested.

Figure 13: Episodes Server module settings form.

## Reports

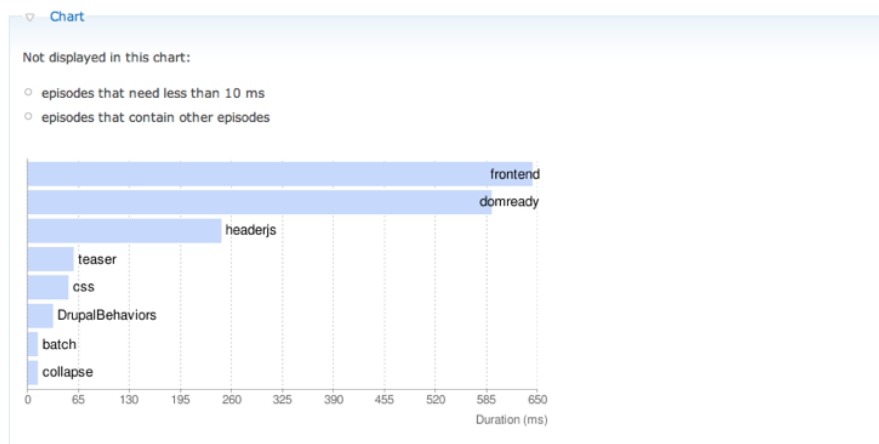
3750 episode measurements have been collected over 379 page views (58 of which also contain measurements of the back-end) since Sunday, March 15, 2009 - 01:15.

### Page loading performance



Table

### Episodes



Table

Figure 14: Report generated by the Episodes Server module.

### 10.4.3 Desired future features

Due to lack of time, the basic reports are ... well ... very basic. It'd be nice to have more charts and to be able to filter the data of the charts. In particular, these 3 filters would be very useful:

1. filter by timespan: all time, 1 year, 6 months, 1 month, 1 week, 1 day
2. filter by browser and browser version
3. filter by (parts of) the URL



## 10.5 Insights

- Episodes module
  - Generating the back-end start time on the server can never work reliably because the clocks of the client (browser) and server are never perfectly in sync, which is required.
  - Even just measuring the page execution time on the server cannot work because of this same reason. You *can* accurately measure this time, but you can't *relate* it to the measurements in the browser. I implemented this using implementations Drupal's `hook_boot()` and `hook_exit()` hooks and came to this conclusion.
  - On the first page load, the `onbeforeunload` cookie is not yet set and therefor the `backend` episode cannot be calculated, which in turn prevents the `pageready` and `totaltime` episodes from being calculated. This is of course also a problem when cookies are disabled, because then the `backend` episode can *never* be calculated. There is no way around this until the day that browsers provide something like `document.requestTime`.
- Episodes Server module
  - Currently the same database as Drupal is being used. Is this scalable enough for analyzing the logs of web sites with millions of page views? No. Writing everything to a SQLite database wouldn't be better. The real solution is to use a different server to log to or even a web service. Better even is to log to a non-app server of your own and then send the logs to an external web service. Then you stay in control of all your data! (Avoiding vendor lock-in.) The main reason I opted for using the same database, is ease of development. Optimizing the profiling tool is not the goal of this bachelor thesis, optimizing page loading performance is.

## 10.6 Feedback from Steve Souders

I explained Steve Souders what I wanted to achieve through this bachelor thesis and the initial work I had already done on integrating Episodes with Drupal. This is how his reply started:

Wow.

Wow, this is awesome.

So, at least he thinks that this was a worthwhile job and if he thinks that, then it will probably be worthwhile/helpful for the Drupal community as well.

## 11 Daemon

So now that we have the tools to accurately (or at least *representatively*) measure the effects of using a CDN, we still have to start using a CDN. So, let's look at how we can make a web site use a CDN.

As explained in section 8 on page 9, there are two very different methods for populating CDNs. Supporting pull is easy, supporting push is a lot of work. But if we want to avoid vendor lock-in, it's necessary to be able to transparently switch between either pull or any of the transfer protocols for push.

And to avoid vendor lock-in even further, it's necessary that we can do the preprocessing ourselves (be that video transcoding, image optimization or anything else).

That's why the meat of this thesis is about a daemon that makes it just as easy to use either push or pull CDNs and that gives you full flexibility in what kind of preprocessing you would like to perform. All you will have to do to integrate your web site with a CDN is:

1. installing the daemon
2. tell it what to do by filling out a simple configuration file
3. start the daemon
4. retrieve the URLs of the synced files from an SQLite database

### 11.1 Goals

As said before, the ability to use either push or pull CDNs is an absolute necessity, as is the ability to process files before they are synced to the CDN. However, there's more to it than just that, so here's a full list of goals.

- Easy to use: the configuration file is the interface and explain itself just by its structure
- Transparency: the transfer protocol(s) supported by the CDN should be irrelevant
- Mixing CDNs and static file servers
- Processing before sync: image optimization, video transcoding ...
- Detect (& sync) new files instantly: through inotify on Linux, FSEvents on Mac OS X and the FindFirstChangeNotification AP or ReadDirectoryChanges API on Windows (there's also the FileSystemWatcher class for .NET)

- Robustness: when the daemon is stopped (or when it crashed), it should know where it left off and sync all added, modified and deleted files
- Scalable: syncing 1,000 or 1,000,000 files – and keeping them synced – should work just as well
- Unit testing wherever feasible
- Design for reuse wherever possible
- Low resource consumption (except for processors, which may be very demanding because of their nature)
- No dependencies other than Python (but processors can have additional dependencies)

A couple of these goals need more explaining.

The transparency goal should speak for itself, but you may not yet have realized its impact: this is what will avoid high CDN provider switching costs, that is, it helps to avoid vendor lock-in.

Detecting and syncing files instantly is a must to ensure CDN usage is as high as possible. If new files would only be detected every 10 minutes, then visitors, who might have uploaded images as part of the content they've created, will be downloading the image from the web server (or maybe static file server), which is suboptimal, considering that they could've been downloading it from the CDN.

The ability to mix CDNs and static file servers makes it possible to either maximize the page loading performance or minimize the costs. Depending on your company's customerbase, you may either want to pay for a global CDN or a local one. If you're a global company, a global CDN makes sense. But if you're present only in a couple of countries, say the U.S.A., Japan and France, it doesn't make sense to pay for a global CDN. It's probably cheaper to pay for a North-American CDN and a couple of strategically placed static file servers in Japan and France to cover the rest of your customer base. Without this daemon, this is rather hard to set up. With it however, it becomes child's play: all you have to do, is configure multiple destinations. That's all there is to it. It is then still up to you how you use these files, though. To decide from which server you will let your visitors download the files, you could look at the IP, or if your visitors must register, at the country they've set in their profile. This also allows for event-driven server allocation. For example if a big event is being hosted in Paris, you could temporarily hire another server in Paris to ensure low latency and high throughput.

#### **Other use cases**

- Back-up tool
- Transcoding server

- Key component in creating your own CDN
- Key component in a file synchronization tool for consumers

## 11.2 Configuration file design

Since the configuration file is the interface and I had a good idea of the features I wanted to support, I started by writing a configuration file. That might be unorthodox, but in the end, this is the most important part of the daemon. If it's too hard to configure, nobody will use it. If it's easy to use, more people will be inclined to give it a try.

Judge for yourself how easy it is by looking at listing 3 on page 37. Beneath the config root node, there are 3 child nodes, each for one of the 3 major sections:

1. **sources**: indicate each data source, in which new, modified and deleted files will be detected recursively. Each source has a name (that we will reference later in the configuration file) and of course a **scanPath**, which defines the root directory within which new/modified/deleted files will be detected. It can also optionally have the **documentRoot** and **basePath** attributes, which may be necessary for some processors that perform magic with URLs.
2. **servers**: provide the settings for all servers that will be used in this configuration. Each server has a **name** and a **transporter** that it should use. The child nodes of the **server** node are the settings that are passed to that transporter.
3. **rules**: this is the heart of the configuration file, since this is what determines what goes where. Each rule is associated with a source (via the **for** attribute), must have a **label** attribute and can consist (but doesn't have to!) of three parts:
  - (a) **filter**: can contain **paths**, **extensions**, **ignoredDirs**, **pattern** and **size** child nodes. The text values of these nodes will be used to filter the files that have been created, modified or deleted within the source to which this rule applies. If it's a match, then the rule will be applied (and therefor the processor chain and destination associated with it). Otherwise, this rule is ignored for that file. See the filter module (section 11.3.1) explanation for details.
  - (b) **processorChain**: accepts any number of **processor** nodes through which you reference (via the **name** attribute) the processor module and the specific processor class within that processor module that you'd like to use. They'll be chained in the order you specify here.
  - (c) **destinations**: accepts any number of **destination** nodes through which you specify all servers to which the file should be transported. Each **destination** node must have a **server** attribute and can have a **path** attribute. The **path** attribute sets a parent path inside which the files will be transported.

Reading the above should make less sense than simply reading the configuration file. If that's the case for you too, then I succeeded.

## 11.3 Python modules

All modules have been written with reusability in mind: none of them make assumptions about the daemon itself and are therefore reusable in other Python applications.

### 11.3.1 filter.py

This module provides the `Filter` class. Through this class, you can check if a given file path matches a set of conditions. This class is used to determine which processors should be applied to a given file and to which CDN it should be synced.

This class has just 2 methods: `set_conditions()` and `matches()`. There are 5 different conditions you can set. The last two should be used with care, because they are a lot slower than the first three. Especially the last one can be very slow, because it must access the file system.

If there are several valid options within a single condition, a match with *any* of them is sufficient (OR). Finally, *all* conditions must be satisfied (AND) before a given file path will result in a positive match.

The five conditions are:

1. `paths`: a list of paths (separated by colons) in which the file can reside
2. `extensions`: a list of extensions (separated by colons) the file can have
3. `ignoredDirs`: a list of directories (separated by colons) that should be ignored, meaning that if the file is inside one of those directories, `Filter` will mark this as a negative match – this is useful to ignore data in typical CVS and `.svn` directories
4. `pattern`: a regular expression the file path must match
5. `size`
  - (a) `conditionType`: either `minimum` or `maximum`
  - (b) `treshold`: the treshold in bytes

This module is fully unit-tested and is therefore guaranteed to work flawlessly.

Listing 3: Sample configuration file.

```

<?xml version="1.0" encoding="UTF-8"?>
<config>
  <!-- Sources -->
  <sources>
    <source name="drupal" scanPath="/htdocs/drupal" documentRoot="/htdocs" basePath="/drupal/" />
    <source name="downloads" scanPath="/Users/wimleers/Downloads" />
  </sources>

  <!-- Servers -->
  <servers>
    <server name="origin pull cdn" transporter="symlink_or_copy">
      <location>/htdocs/drupal/staticfiles</location>
      <url>http://mydomain.mycdn.com/staticfiles</url>
    </server>
    <server name="ftp push cdn" transporter="ftp" maxConnections="5">
      <host>localhost</host>
      <username>daemontest</username>
      <password>daemontest</password>
      <url>http://localhost/daemontest/</url>
    </server>
  </servers>

  <!-- Rules -->
  <rules>
    <rule for="drupal" label="CSS, JS, images and Flash">
      <filter>
        <paths>modules:misc</paths>
        <extensions>ico:js:css:gif:png:jpg:jpeg:svg:swf</extensions>
        <ignoredDirs>CVS:.svn</ignoredDirs>
      </filter>
      <processorChain>
        <processor name="image_optimizer.KeepFilename" />
        <processor name="yui_compressor.YUICompressor" />
        <processor name="link_updater.CSSURLUpdater" />
        <processor name="unique_filename.Mtime" />
      </processorChain>
      <destinations>
        <destination server="origin pull cdn" />
        <destination server="ftp push cdn" path="static" />
      </destinations>
    </rule>

    <rule for="drupal" label="Videos">
      <filter>
        <paths>modules:misc</paths>
        <extensions>flv:mov:avi:wmv</extensions>
        <ignoredDirs>CVS:.svn</ignoredDirs>
        <size conditionType="minimum">1000000</size>
      </filter>
      <processorChain>
        <processor name="unique_filename.MD5" />
      </processorChain>
      <destinations>
        <destination server="ftp push cdn" path="videos" />
      </destinations>
    </rule>

    <rule for="downloads" label="Mirror">
      <filter>
        <extensions>mov:avi</extensions>
      </filter>
      <destinations>
        <destination server="origin pull cdn" path="mirror" />
        <destination server="ftp push cdn" path="mirror" />
      </destinations>
    </rule>
  </rules>
</config>

```

### 11.3.2 pathscanner.py

As is to be expected, this module provides the `PathScanner` class, which scans paths and stores them in a SQLite [54] database. You can use `PathScanner` to detect changes in a directory structure. For efficiency, only creations, deletions and modifications are detected, not moves. This class is used to scan the file system for changes when no supported filesystem monitor is installed on the current operating system. It is also used for persistent storage: when the daemon has been stopped, the database built and maintained through/by this class is used as a reference, to detect changes that have happened before it was started again.

The database schema is very simple: (`path`, `filename`, `mtime`). Directories are also stored; in that case, `path` is the path of the parent directory, `filename` is the directory name and `mtime` is set to -1. Modified files are detected by looking comparing the current `mtime` with the value stored in the `mtime` column.

Changes to the database are committed in batches, because changes in the filesystem typically occur in batches as well. If every change would be committed separately, the concurrency level would rise unnecessarily. By default, every batch of 50 changes are committed.

This class provides you with 8 methods:

- `initial_scan()` to build the initial database – works recursively
- `scan()` to get the changes – doesn't work recursively
- `scan_tree()` (uses `scan()`) to get the changes in an entire directory structure – obviously works recursively
- `purge_path()` to purge all the metadata for a path from the database
- `add_files()`, `update_files()`, `remove_files()` to add/update/remove files manually (useful when your application has more/faster knowledge of changes)

This module doesn't have any tests yet, because it requires *a lot* of mock functions to simulate system calls. It's been tested manually thoroughly though.

### 11.3.3 fsmonitor.py

This time around, there's more to it than it seems. `fsmonitor.py` provides `FSMonitor`, a base class from which subclasses derive. `fsmonitor_inotify.py` has the `FSMonitorInotify` class, `fsmonitor_fsevents.py` has `FSMonitorFSEvents` and `fsmonitor_polling.py` has `FSMonitorPolling`. Put these together, and you have a single, simple to use abstraction to use each operating system's file system monitor:

- Uses inotify [49] on Linux (kernel 2.6.13 and higher)
- Uses FSEvents [51, 52] on Mac OS X (10.5 and higher)
- Falls back to polling when neither one is present

Windows support is possible, but hasn't been implemented yet due to time constraints.

### Implementation obstacles

To make this class work consistently, less critical features that are only available for specific file system monitors are abstracted away. And other features are emulated. It comes down to the fact that `FSMonitor`'s API is very simple to use and only supports 5 different events: `CREATED`, `MODIFIED`, `DELETED`, `MONITORED_DIR_MOVED` and `DROPPED_EVENTS`. The last 2 events are only triggered for inotify and FSEvents.

A persistent mode is also supported, in which all metadata is stored in a database. This allows you to even track changes when your program wasn't running.

As you can see, only 3 “real” events of interest are supported: the most common ones. This is because not every API supports all features of the other APIs. inotify is the most complete in this regard: it supports a boatload of different events, on a file-level. But it only provides realtime events: it doesn't maintain a complete history of events. And that's understandable: it's impossible to maintain a history of *every* file system event. Over time, there wouldn't be any space left to store actual data.

FSEvents on the other hand, works on the directory-level, so you have to maintain your own directory tree state to detect created, modified and deleted files. It only triggers a “a change has occurred in this directory” event. This is why for example there is no “file moved” event: it'd be too resource-intensive to detect that, or at the very least it wouldn't scale. On the plus side, FSEvents maintains a complete history of events.

Implementations that don't support file-level events (FSEvents and polling) are persistent by design. Because the directory tree state must be maintained to be able to trigger the correct events, `PathScanner` (see section 11.3.2) is used for storage. They use `PathScanner`'s `(add|update|remove)_files()` functions to keep the database up-to-date. And because the entire directory tree state is always stored, you can compare the current directory tree state with the stored one to detect changes, either as they occur (by being notified of changes on the directory level) or as they have occurred (by scanning the directory tree manually).

For the persistent mode, we could take advantage of `FSEvents`' ability to look back in time. However, again due to time constraints, the same approach is used for every implementation: a manual scanning procedure – using `PathScanner` – is started after the file system monitor starts listening for new events on a given path. That way, no new events are missed. This works equally well as using `FSEvents`' special support for this: it's just slower. But it's sufficient for now.



### 11.3.4 `persistent_queue.py` and `persistent_list.py`

In order to provide data persistency, I wrote the `PersistentQueue` and `PersistentList` classes. As their names indicate, these provide you with a persistent queue and a persistent list. They use again an SQLite database for persistent storage. For each instance you create, you must choose the table name and you can optionally choose which database file to write to. This allows you to group persistent datastructures in a logical manner (i.e. related persistent datastructures can be stored in the same database file, thereby also making it portable and easy to backup).

To prevent excessive file system access due to an overreliance on SQLite, I also added in-memory caching. To ensure low resource consumption, only the first `X` items in `PersistentQueue` are cached in-memory (a minimum and maximum treshold can be configured), but for `PersistentList` there is no such restriction: it's cached in-memory in its entirety. It's not designed for large datasets, but `PersistentQueue` is.

The first question to arise is: “Why use SQLite in favor of Python's built-in `shelve` module?” Well, the answer is simple: aside of the benefit of the ability to have all persistent data in a single file, it must also scale to tens of thousands or even millions of files. `shelve` isn't scalable because its data is loaded into memory in its entirety. This could easily result in hundreds of megabytes of memory usage. Such excessive memory usage should be avoided at all costs when the target environment is a (web) server.

Your next question would probably be: “How can you be sure the SQLite database won't get corrupt?” The answer is: we can't. But the same applies to Python's `shelve` module. However, the aforementioned advantages of SQLite give plenty of advantages over `shelve`. Plus, SQLite is thoroughly tested, even against corruption [55]. It's also used for very large datasets and by countless companies. So it's the best bet you can make.

Finally, you'd probably ask “Why not use MySQL or PostgreSQL or ...?”. Again the answer is brief: because SQLite requires no additional setup because it is *serverless*, as opposed to MySQL and PostgreSQL.

Both modules are fully unit-tested and are therefor guaranteed to work flawlessly.

### 11.3.5 Processors

#### `processor.py`

This module provides several classes: `Processor`, `ProcessorChain` and `ProcessorChainFactory`.

`Processor` is a base class for processors, which are designed to be easy to write yourself. Processors receive an input file, do something with it and return the

Listing 4: YUICompressor Processor class.

```
class YUICompressor(Processor):
    """compresses .css and .js files with YUI Compressor"""

    valid_extensions = (".css", ".js")

    def run(self):
        # We don't rename the file, so we can use the default output file.

        # Remove the output file if it already exists, otherwise YUI
        # Compressor will fail.
        if os.path.exists(self.output_file):
            os.remove(self.output_file)

        # Run YUI Compressor on the file.
        yuicompressor_path = os.path.join(self.processors_path, "yuicompressor.jar")
        args = (yuicompressor_path, self.input_file, self.output_file)
        (stdout, stderr) = self.run_command("java -jar %s %s -o %s" % args)

        # Raise an exception if an error occurred.
        if not stderr == "":
            raise ProcessorError(stderr)

        return self.output_file
```

output file. The `Processor` class takes a lot of the small annoying tasks on its shoulders, such as checking if the file is actually a file this processor can process, calculating the default output file and a simple abstraction around an otherwise multi-line construction to run a command.

Upon completion, a callback will be called. Another callback is called in case of an error.

An example can be found in listing 4. For details, please consult the daemon's documentation.

Processors are allowed to make any change they want to the file's contents. They are also allowed to change the base name of the input file, but they're not allowed to change its path. This measure was taken to reduce the amount of data that will need to be stored to know which file is stored where exactly. This is enforced *by convention*, because in Python it's impossible to truly enforce anything. If you do change the path, the file will sync just fine, but it will be impossible to delete the old version of a modified file, unless it results in the exact same path and base name each time it runs through the processor chain.

The `Processor` class accepts a parent logger which subclasses can optionally use to perform logging.

Then there is the `ProcessorChain` class, which receives a list of processors and then runs them as a chain: the output file of processor 1 is the input file of processor 2, and so on. `ProcessorChains` run in their own threads. `ProcessorChain` also supports logging and accepts a parent logger.

There are two special exceptions `Processor` subclasses can throw:

1. `RequestToRequeueException`: when raised, the `ProcessorChain` will stop processing this file and will pretend the processing failed. This effectively

means that the file will be reprocessed later. A sample use case is the `CSSURLUpdater` class [11.3.5](#), in which the URLs of a CSS file must be updated to point to the corresponding URLs of the files on the CDN. But if not all of these files have been synced, that's impossible. So it must be retried later.

2. `DocumentRootAndBasePathRequiredException`: when raised, the `ProcessorChain` will stop applying the processor that raised this exception to this file, because the source to which this file belongs, did not have these attributes set and therefore it cannot be applied.

Finally, `ProcessorChainFactory`: this is simply a factory that generates `ProcessorChain` objects, with some parameters already filled out.

### **filename.py**

This processor module provides two processor classes: `SpacesToUnderscores` and `SpacesToDashes`. They respectively replace spaces with underscores and spaces with dashes in the base name of the file.

This one is not very useful, but it's a good simple example.

### **unique\_filename.py**

Also in this processor module, two processor classes are provided: `Mtime` and `MD5`. `MTime` appends the mtime (last modification time) as a UNIX timestamp to the file's base name (preceded by an underscore). `MD5` does the same, but instead of the mtime, it appends the MD5 hash of the file to the file's base name.

This processor is useful if you want to ensure that files have unique filenames, so that they can be given far future Expires headers (see section [7 on page 8](#)).

### **image\_optimizer.py**

This processor module is inspired by [\[58\]](#). It optimizes images losslessly, i.e. it reduces the filesize without touching the quality. The research necessary wasn't performed by me, but by Stoyan Stefanov, a Yahoo! web developer working for the Exceptional Performance team, and was thoroughly laid out in a series of blog posts [\[59, 60, 61, 62\]](#) at the Yahoo! user interface blog.

For GIF files, a conversion to PNG8 is performed using ImageMagick's [\[63\]](#) `convert`. PNG8 offers lossless image quality, as does GIF, but results in a smaller file size. PNG8 is also supported in all browsers, including IE6. It's the alpha channels of truecolor PNG (PNG24 & PNG32) that are not supported in IE6.

PNG files are stored in so-called “chunks” and not all of these are required to display the image – in fact, most of them are not used at all. `pngcrush` [64] is used to strip all the unneeded chunks. `pngcrush` is also applied to the PNG8 files that are generated by the previous step. I decided not to use the brute force method, which tries over a hundred different methods for optimization, but just the 10 most common ones. The brute force method would result in 30 seconds of processing versus less than a second otherwise.

JPEG files can be optimized in three complementary ways: stripping metadata, optimizing the Huffman tables and making them progressive. There are two variations to store a JPEG file: baseline and progressive. A baseline JPEG file is stored as one top-to-bottom scan, whereas a progressive JPEG file is stored as a series of scans, with each scan gradually improving the quality of the overall image. Stoyan Stefanov’s tests [62] have pointed out that you’ve got a 75% chance that the JPEG file is best saved as baseline when it’s smaller than 10 KB. For JPEG files larger than 10 KB, it’s 94% likely that progressive JPEG will result in a better compression ratio. That’s why the third optimization (making JPEG files progressive) is only applied when the file is larger than 10 KB. All these optimizations are applied using `jpegtran` [65].

Finally, animated GIF files can be optimized by stripping the pixels from each frame that don’t change from the previous to the next frame. I use `gifsicle` [66] to achieve that.

One nuance you should know about: stripping metadata may also remove the copyright information, which may have legal consequences. So you better don’t strip metadata when you’ve bought some of the photos you’re hosting, which may be the case for e.g. a newspaper web site.

Now that you know how the optimizations are done, here is the overview of all processor classes that this processor module provides:

1. `Max` optimizes image files losslessly (GIF, PNG, JPEG, animated GIF)
2. `KeepMetadata` same as `Max`, but keeps JPEG metadata
3. `KeepFilename` same as `Max`, but keeps the original filename (no GIF optimization)
4. `KeepMetadataAndFilename` same as `Max`, but keeps JPEG metadata and the original filename (no GIF optimization)

### **link\_updater.py**

Thanks to this processor module, it is possible to serve CSS files from a CDN while updating the URLs in the CSS file to reference the new URLs of these files, that is, the URLs of the synced files. It provides a sole processor class: `CSSURLUpdater`. This processor class should *only* be used when *either* of these conditions are true:

- The base path of the URLs changes and relative URLs that are relative to the document root.  
For example, `http://example.com/static/css/style.css` becomes `http://cdn.com/example.com/` and its referenced file `http://example.com/static/images/background.png` becomes `http://cdn.com/example.com/static/images/background.png` after syncing. If `style.css` referenced `background.png` through the relative URL `/static/images/background.png`, then it must use `CSSURLUpdater`. Otherwise this relative URL would become invalid.
- The base names of the referenced files changes.  
For example, `http://example.com/static/css/style.css` becomes `http://cdn.com/example.com/` and its referenced file `http://example.com/static/images/background.png` becomes `http://cdn.com/static/images/background_1242440827.png` after syncing. Then it must always use `CSSURLUpdater`. Otherwise the URL would become invalid, as the file's base name has changed.

`CSSURLUpdater` uses the `cssutils` [67] Python module to parse CSS files. This unfortunately also negatively impacts its performance, because it validates the CSS file while tokenizing it. But as this will become open source, others will surely improve this. A possibility is to use regular expressions instead to filter out the URLs.

All `CSSURLUpdater` does is resolving relative URLs (relative to the CSS file or relative to the document root) to absolute paths on the file system, then looking up the corresponding URLs on the CDN and placing those instead in the CSS file. If one of the referenced files cannot be found on the file system, this URL remains unchanged. If one of the referenced files has not yet been synced to the CDN, then a `RequestToRequeueException` exception will be raised (see section 11.3.5 on page 40) so that another attempt will be made later, when hopefully all referenced files have been synced. For details, see the daemon's documentation.

### **yui\_compressor.py**

This is the processor module that could be seen in listing 4 on page 41. It accepts CSS and JS files and runs the YUI Compressor [68] on them, which are then compressed by stripping out all whitespace and comments. For JavaScript, it relies on Rhino [69] to tokenize the JavaScript source, so it's very safe: it won't strip out whitespace where that could potentially cause problems. Thanks to this, it can also optimize more aggressively: it saves over 20% more than JSMIN [70]. For CSS (which is supported since version 2.0) it uses a regular-expression based CSS minifier.

Listing 5: `TransporterFTP` `Transporter` class.

```
class TransporterFTP(Transporter):
    name = 'FTP'
    valid_settings = ImmutableSet(["host", "username", "password", "url", "port", "path"])
    required_settings = ImmutableSet(["host", "username", "password", "url"])

    def __init__(self, settings, callback, error_callback, parent_logger=None):
        Transporter.__init__(self, settings, callback, error_callback, parent_logger)

        # Fill out defaults if necessary.
        configured_settings = Set(self.settings.keys())
        if not "port" in configured_settings:
            self.settings["port"] = 21
        if not "path" in configured_settings:
            self.settings["path"] = ""

        # Map the settings to the format expected by FTPStorage.
        location = "ftp://" + self.settings["username"] + ":"
        location += self.settings["password"] + "@" + self.settings["host"]
        location += ":" + str(self.settings["port"]) + self.settings["path"]
        self.storage = FTPStorage(location, self.settings["url"])
        try:
            self.storage._start_connection()
        except Exception, e:
            raise ConnectionError(e)
```

### 11.3.6 Transporters

#### `transporter.py`

Each transporter is a persistent connection to a server via a certain protocol (FTP, SCP, SSH, or custom protocols such as Amazon S3, any protocol really) that is running in its own thread. It allows you to queue files to be synced (save or delete) to the server.

`Transporter` is a base class for transporters, which are in turn very (very!) thin wrappers around custom Django storage systems [72]. If you need support for another storage system, you should write a custom Django storage system first. Transporters' settings are automatically validated in the constructor. Also in the constructor, an attempt is made to set up a connection to their target server. When that fails, an exception (`ConnectionError`) is raised. Files can be queued for synchronization through the `sync_file(src, dst, action, callback, error_callback)` method.

Upon completion, the `callback` function will be called. The `error_callback` function is called in case of an error.

`Transporter` also supports logging and accepts a parent logger.

A sample transporter can be found in listing 5. For details, please consult the daemon's documentation.

Now, why the dependency on Django's `Storage` class? For three reasons:

1. Since Django is a pretty big open source project with many developers and is powering many web sites, it's fair to assume that the API is stable and solid. Reinventing the wheel is meaningless and will just introduce more bugs.

2. Because the daemon relies on (unmodified!) Django code, it can benefit from bugfixes/features applied to Django's code and can use custom storage systems written for Django. The opposite is also true: changes made by contributors to the daemon (and initially myself) can be contributed back to Django and its contributed custom storage systems.
3. `django-storages` [73] is a collection of custom storage systems, which includes these classes:
  - (a) `DatabaseStorage`: store files in the database (any database that Django supports (MySQL, PostgreSQL, SQLite and Oracle))
  - (b) `MogileFSStorage`; MogileFS [81] is an open source distributed file system
  - (c) `CouchDBStorage`; Apache CouchDB [82] is a distributed, fault-tolerant and schema-free document-oriented database accessible via a RESTful HTTP/JSON API.
  - (d) `S3Storage`; uses the official Amazon S3 Python module
  - (e) `S3BotoStorage`; uses the boto [80] module to access Amazon S3 [78] and Amazon CloudFront [79]
  - (f) `FTPStorage`; uses the `ftplib` Python module [77]

For the last two, transporters are available. The first three are not so widely used and thus not yet implemented, although it'd be very easy to support them, exactly because all that is necessary, is to write thin wrappers. The fourth is not very meaningful to use, since the fifth is better (better maintained and better performing).

So I clearly managed to make a big shortcut (for simplicity of the argument, I'm waving away the fact that I had to get this to work outside of Django itself) to achieve my goal: support CDNs that support FTP or origin pulling (see section 8 on page 9), as well as the Amazon S3 and CloudFront CDNs.

However, supporting origin pull was trickier than would seem at first. Normally, you just rewrite your URLs and be done with it. However, I wanted to support processing files prior to syncing them to the CDN. And I want to keep following the "don't touch the original file" rule. With push, that is no problem, you just process the file, store the output file in a temporary directory, push the file and delete it afterwards. But what about pull?

I had to be creative here. Since files must remain available for origin pull (in case the CDN wants/needs to update its copy), all files must be copied to another publicly accessible path in the web site. But what about files that are not modified? Or have just changed filenames (for unique URLs)? Copying these means storing the exact same data twice. The answer is fortunately very simple: symlinks. Although available only on UNIX, it's very much worth it: it reduces redundant data storage significantly. This was then implemented in a new custom storage system: `SymLinkOrCopyStorage`, which copies modified files and symlinks unmodified ones.

In total, I've contributed three patches to `django-storages`:

1. **FTPStorage**: saving large files + more robust `exists()` [74]

- (a) It enables the saving of large files by no longer reading all the chunks of the file in a single string, instead it uses `ftplib.storbinary()` directly with a file pointer, which then handles the writing in chunks automatically.
- (b) It makes `exists()` more reliable: it's been tested with 3 different FTP servers and so far it works without problems with the following FTP servers, whereas it didn't work with any of them before:
  - i. PureFTPd
  - ii. Xlight FTP Server 3.2 (used by SimpleCDN)
  - iii. Pure-FTPd (used by Rambla)

This greatly improves the number of use cases where you can use the **FTPStorage** custom storage system.

2. **S3BotoStorage**: set `Content-Type` header, ACL fixed, use HTTP and disable query auth by default [75]

- (a) The `Content-Type` header is set automatically via guessing based on the extension, this is done through `mimetypes.guess_type`. Right now, no `Content-Type` is set, and therefore the default binary mimetype is set: `application/octet-stream`. This causes browsers to download files instead of displaying them.
- (b) The ACL now actually gets applied properly to the bucket and to each file that is saved to the bucket.
- (c) Currently, URLs are generated with query-based authentication (meaning you'll get ridiculously long URLs) and HTTPS is used instead of HTTP, thereby preventing browsers from caching files. I've disabled query authentication and HTTPS, as this is the most common use case for serving files. This probably should be configurable, but that can be done in a revised patch or a follow-up patch.
- (d) It allows you to set custom headers through the constructor (which I really needed for my daemon).

This greatly improves the usability of the **S3BotoStorage** custom storage system in its most common use case: as a CDN for publicly accessible files.

3. **SymLinkOrCopyStorage**: new custom storage system [76]

The maintainer was very receptive to these patches and replied a mere 23 minutes after I contacted him (via Twitter):

dauidbgk@wimleers Impressive patches, I'll merge your work asap.  
Thanks for contributing! Interesting bachelor thesis :)



The patches were submitted on May 14, 2009. The first and third patch were committed on May 17, 2009. The second patch needs a bit more work (more configurable, less hard coded, which it already was though). The fact that another new storage system was also added (Apache CouchDB) seems to indicate that I made a good choice: this project seems to be pretty active and is gaining attention.

#### **transporter\_ftp.py**

Provides the `TransporterFTP` class, which is a thin wrapper around `FTPStorage`, with the aforementioned patch applied.

#### **transporter\_s3.py**

Provides the `TransporterS3` class, which is a thin wrapper around `S3BotoStorage`, with the aforementioned patch applied.

#### **transporter\_cf.py**

Provides the `TransporterCF` class, which is not a thin wrapper around `S3BotoStorage`, but around `TransporterS3`. In fact, it just implements the `alter_url()` method to alter the Amazon S3 URL to an Amazon CloudFront URL .

It also provides the `create_distribution()` function to create a distribution for a given origin domain (a domain for a specific Amazon S3 bucket). Please consult the daemon's documentation for details.

#### **transporter\_symlink\_or\_copy.py**

Provides the `TransporterSymlinkOrCopy` class, which is a thin wrapper around `SymlinkOrCopyStorage`, which is a new custom storage system I contributed to `django-storages`, as mentioned before.

#### **11.3.7 config.py**

This module contains just one class: `Config`. `Config` can load a configuration file (parse the XML) and validate it. Validation doesn't happen through an XML schema, but through "manual" validation. The `filter` node is validated through the `Filter` class to ensure it is error free. All references (to sources and servers) are also validated. Its validation routines are pretty thorough, but by no means perfect.

`Config` also supports logging and accepts a parent logger.

This module should be unit tested, but isn't – yet.

### 11.3.8 daemon\_thread\_runner.py

I needed to be able to run the application as a daemon. Great, but then how do you stop it? Through signals. That's also how for example the Apache HTTP server does it [83]. To send a signal, you need to know the process' pid (process id). So the pid must be stored in a file somewhere.

This module contains the `DaemonThreadRunner` class, which accepts an object and the name of the file that should contain the pid. The object should be a subclass of Python's `threading.Thread` class. As soon as you `start()` the `DaemonThreadRunner` object, the pid will be written to the specified pid file name, the object will be marked as a daemon thread and started. While it's running, the pid is written to the pid file every 60 seconds.

When an interrupt is caught (`SIGINT` for interruption, `SIGTSTP` for suspension and `SIGTERM` for termination), the thread (of the object that was passed) is stopped and `DaemonThreadRunner` waits for the thread to join and then deletes the file.

This module is not unit tested, because it makes very little sense to do so (there's not much code). Having used it hundreds of times, it didn't fail once, so it's reliable enough.

## 11.4 Putting it all together: arbitrator.py

### 11.4.1 The big picture

The arbitrator is what links together all Python modules I've described in the previous section. Here's a hierarchical overview, so you get a better understanding of The big picture:

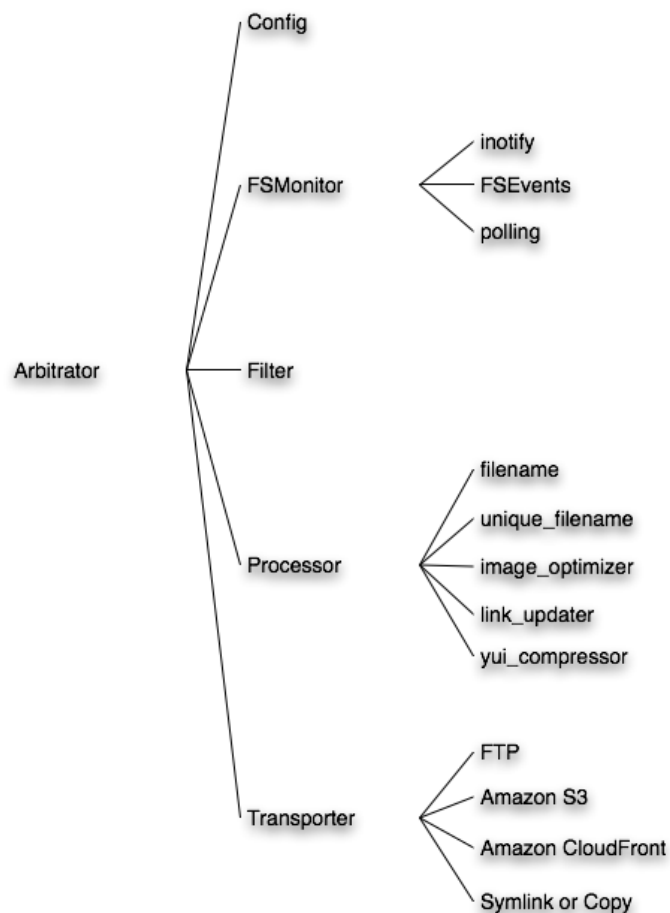


Figure 15: The big picture

Clearly, `Arbitrator` is what links everything together: it controls the 5 components: `Config`, `FSMonitor`, `Filter`, `Processor` and `Transporter`. There are three subclasses of `FSMonitor` to take advantage of the platform's built-in file system monitor. `Processor` must be subclassed for every processor. `Transporter` must be subclassed for each protocol.

Now that you have an insight in the big picture, let's see how exactly `Arbitrator` controls all components, and what happens before the `main` function.

#### 11.4.2 The flow

First, an `Arbitrator` object is created and its constructor does the following:

- create a logger

- parse the configuration file
- verify the existence of all processors and transporters that are referenced from the configuration file
- connect to each server (as defined in the configuration file) to ensure it's working

Then, the `Arbitrator` object is passed to a `DaemonThreadRunner` object, which then runs the arbitrator in such a way that it can be stopped through signals. The arbitrator is then started. The following happens:

#### 1. setup

- create transporter pools (cfr. worker thread pools) for each server. These pools remain empty because they're filled
- collect all metadata for each rule
- initialize all datastructures for the pipeline (queues, persistent queues and persistent lists)
- move files from the 'files in pipeline' persistent list to the 'pipeline' persistent queue
- move files from the 'failed files' persistent list to the 'pipeline' persistent queue
- create a database connection to the 'synced files' database
- initialize the file system monitor (`FSMonitor`)

#### 2. run

- start the file system monitor
- start the processing loop and keep it running until the thread is being stopped
  - process the *discover* queue
  - process the *pipeline* queue
  - process the *filter* queue
  - process the *process* queue
  - process the *transport* queues (1 per server)
  - process the *db* queue
  - process the *retry* queue
  - allow retry (move files from the 'failed files' persistent list to the 'pipeline' persistent queue)
  - sleep 0.2 seconds
- stop the file system monitor
- process the discover queue once more to sync the final batch of files to the persistent pipeline queue
- stop all transporters

(f) log some statistics

That's *roughly* the logic of the daemon. It should already make sense to you, but you're probably wondering what all the queues are for. And how they're being filled and emptied. So now it's time to learn about the daemon's *pipeline*.

### 11.4.3 Pipeline design pattern

This design pattern, which is also sometimes called “Filters and Pipes” [84, 85, 86, 87], is slightly underdocumented, but it's still a very useful design pattern. Its premise is to deliver an architecture to divide a large processing task into smaller, sequential steps (“Filters”) that can be performed independently – and therefor in parallel – which are finally connected via Pipes. The output of one step is the input of the next.

For all that follows in this subsection, you may want to look at figure 16 while reading. Note that this figure doesn't contain every detail: it is intended to help you gain some insight into how the daemon works, not how every detail is implemented.

In my case, files are discovered and are then put into the pipeline queue. When they actually move into the pipeline (at which point they're added to the 'files in pipeline' persistent list), they start by going into the filter queue, after being filtered they go into the process queue (possibly more than once), after being processed to the transport queue (again possibly more than once), after being transported to the db queue, after being stored in the database, they're removed from the 'files in pipeline' persistent list and we're done for this file. Repeat for every discovered file. This is the *core logic* of the daemon.

So many queues are used because there are so many stages in the pipeline. There's a queue for each stage in the pipeline, plus some additional ones because the persistent datastructures use the pysqlite module, which only allows you to access the database from the same thread as the connection was created in. Because I (have to) work with callbacks, the calling thread may be different from the creating thread, and therefor there are several queues that exist solely for exchanging data between threads.

There is one persistent queue and two persistent lists. The persistent queue is the pipeline queue, which contains all files that are queued to be sent through the pipeline. The first persistent list is 'files in pipeline'. It is used to ensure files still get processed if the daemon was killed (or crashed) while they were in the pipeline. The second persistent list is 'failed files' and contains all files for which either a processor in the processor chain or a transporter failed.

When the daemon is restarted, the contents of the 'files in pipeline' and 'failed files' lists are pushed into the pipeline queue, after which they are erased.

Queues are either filled through the **Arbitrator** (because it moves data from one queue to the next):

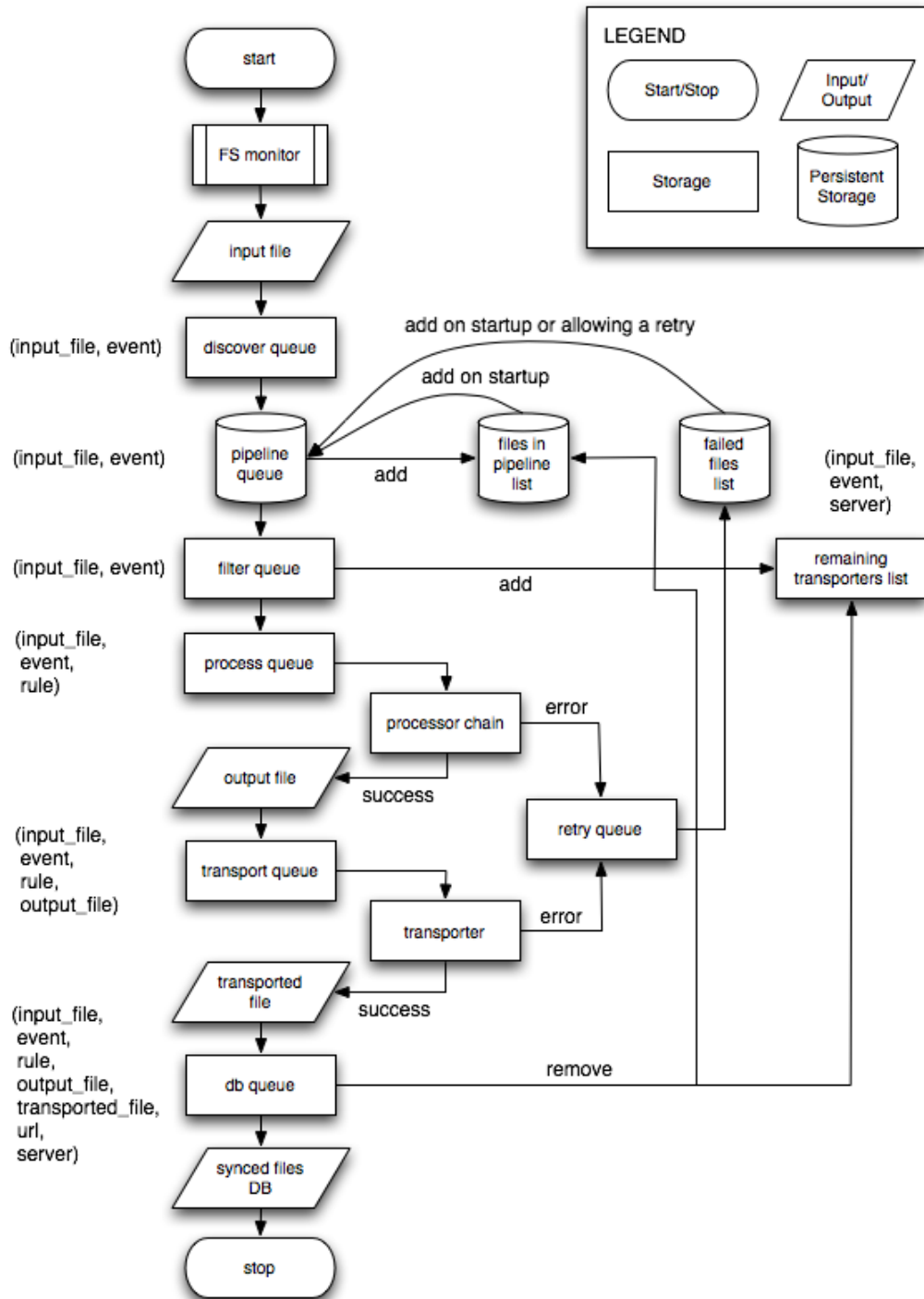


Figure 16: Flowchart of the daemon's pipeline.

- The *pipeline* queue is filled by the “process *discover* queue” method, which always syncs all files in the discover queue to the pipeline queue.
- The *filter* queue is filled by the “process *pipeline* queue” method, which processes up to 20 files (this is configurable) in one run, or until there are 100 files in the pipeline (this is also configurable), whichever limit is hit first.
- The *process* queue is filled by the “process *filter* queue” method, which processes up to 20 files in one run.

or through callbacks (in case data gets processed in a separate thread):

- The *discover* queue is filled through `FSMonitor`’s callback (which gets called for every discovered file).
- The *transport* queue is filled through a `ProcessorChain`’s callback or directly from the “process *filter* queue” method (if the rule has no processor chain associated with it). To know when a file has been synced to all its destinations, the ‘remaining transporters’ list gets a new key (the concatenation of the input file, the event and the string representation of the rule) and the value of that key is a list of all servers to which this file will be synced.
- The *db* queue is filled through a `Transporter`’s callback. Each time this callback fires, it also carries information on which server the file has just been transported to. This server is then removed from the ‘remaining transporters’ list for this file. When no servers are left in this list, the sync is complete and the file can be removed from the ‘files in pipeline’ persistent list.

Because the `ProcessorChain` and `Transporter` callbacks only carry information about the file they’ve just been operating on, I had to find an elegant method to transfer the additional metadata for this file, which is necessary to let the file continue through the pipeline. I’ve found this in the form of currying [88]. Currying is dynamically creating a new function that calls another function, but with some arguments already filled out. An example:

```
curried_callback = curry(self.processor_chain_callback, event=event, rule=rule)
```

The function `self.processor_chain_callback` accepts the `event` and `rule` arguments, but the `ProcessorChain` class has no way of accepting “additional data” arguments. So instead of rewriting `ProcessorChain` (and the exact same thing applies to `Transporter`), I simply create a curried callback, that will automatically fill out the arguments that the `ProcessorChain` callback by itself could never fill out.

Each of the “process *X* queue” methods acquires `Arbitrator`’s lock before accessing any of the queues. *Before* a file is removed from the pipeline queue,

it is added to the 'files in pipeline' persistent list (this is possible thanks to `PersistentQueue`'s `peek()` method), and then it is removed from the pipeline queue. This implies that at no time after the file has been added to the pipeline queue, it can be lost. The worst case scenario is that the daemon crashes between adding the file to the 'files in pipeline' persistent list and removing it from the pipeline queue. Then it'll end up twice in the queue. But the second sync will just overwrite the first one, so all that is lost, is CPU time.

The "allow retry" method allows failed files (in the 'failed files' persistent list) to be retried, by adding them back to the pipeline queue. This happens whenever the pipeline queue is getting empty, or every 30 seconds. This ensures processors that use the `RequestToRequeueException` exception can retry.

The only truly weak link is unavoidable: if the daemon crashes somewhere between having performed the callback from `FSMonitor`, adding that file to the discover queue and syncing the file from the discover queue to the pipeline queue (which is necessary due to the thread locality restriction of `pysqlite`).

## 11.5 Performance tests

I've performed fairly extensive tests in both Mac OS X and Linux. The application behaved identically on both platforms, despite the fact that different file system monitors are being used in the background. The rest of this problemless cross-platform functioning is thanks to Python.

All tests were performed on the local network, i.e. with a FTP server running on the localhost. Very small scale tests have been performed with the Amazon S3 and CloudFront transporters, and since they worked, the results should apply to those as well. It doesn't and shouldn't matter which transporter is being used.

At all times, the memory usage remained below 17 MB on Mac OS X and below 7 MB on Linux (unless the `update_linker` processor module was used, in which case it leaks memory like a madman – the `cssutils` Python module is to blame). A backlog of more than 10,000 files was no problem. Synchronizing 10 GB of files was no problem. I also tried a lot of variations in the configuration and all of them worked (well, sometimes it needed some bugfixing of course). Further testing should happen in real-world environments. Even tests in which I forced processors or transporters to crash were completed successfully: no files were lost and they would be synced again after restarting the daemon.

## 11.6 Possible further optimizations

- Files should be moved from the discover queue to the pipeline queue in a separate thread, to minimize the risk of losing files due to a crashed application before files are moved to the pipeline queue. In fact, the discover queue could be eliminated altogether thanks to this.



- Track progress of transporters and allow them to be stopped while still syncing a file.
- Make processors more error resistant by allowing them to check the environment, so they can ensure 3rd party applications, such as YUI Compressor or jpegtran are installed.
- Figure out an automated way of ensuring the correct operating of processors, since they are most likely the cause of problems thanks to the fact that users can easily write their own Processors.
- Automatically copy the synced files DB every X seconds, to prevent long delays for read-only clients. This will only matter on sites where uploads happen more than once per second or so.
- Reorganize code: make a proper packaged structure.
- Make the code redistributable. As a Python egg, or maybe even as binaries for each supported platform.
- Automatically stop transporters after a period of idle time.

## 11.7 Desired future features

- Polling the daemon for its current status (number of files in the queue, files in the pipeline, processors running, transporters running, etc.)
- Support for Munin/Nagios for monitoring (strongly related to the previous feature)
- Ability to limit network usage by more than just the number of connections: also by throughput.
- Ability to limit CPU usage by more than just the number of simultaneous processors.
- Store characteristics of the operations, such as TTS (Time-To-Sync), so that you can analyze this data to configure the daemon to better suit your needs.
- Allow server-specific processor chains (i.e. run the processor chain once for each server the file will be synced to). This allows you to have CSS files that contain URLs rewritten to that specific server. Right now, you can only have *one* processor chain and therefor the CSS file will always reference the same URLs, which may refer to another server.
- Cache the latest configuration file and compare with the new one. If changes occurred to any of the rules, it should detect them on its own and do the necessary resyncing.

## 12 Improving Drupal: CDN integration

It should be obvious by now that we still need a module to integrate Drupal with a CDN, as Drupal doesn't provide such functionality on its own – if it did, then this bachelor thesis would be titled differently. This is the end of the long journey towards supporting the simplest and the most complex CDN or static file server setups one can make. Fortunately, this is all fairly trivial, except for maybe the necessary Drupal core patch.

### 12.1 Goals

The daemon I wrote is not necessary for Origin Pull CDNs. So this module should support those through a simple UI. On the other hand, we must also make it easy to use the daemon in a Drupal site. Let's call the former *basic mode* and the latter *advanced mode*, thereby indicating that the latter is more complex to set up (i.e. it requires you to set up the daemon). So, let's look at the goals:

- shared functionality
  - ability to show per-page statistics: number of files on the page, number of files served from the CDN
  - status report shows if CDN integration is active and displays as an warning if it is disabled or in debug mode (to stress the importance of having it enabled)
- basic mode
  - enter the CDN URL and it'll be used in file URLs automatically
  - ability to only use the CDN for files with certain extensions
- advanced mode
  - enter the absolute path to the synced files database and then file URLs will be looked up from there automatically
  - status report: check if daemon is running, if not, display the report as an error
  - status report: number of synced files, number of files in the pipeline, number of files waiting to enter the pipeline
  - per-page statistics: show from which destination the file is being served
  - per-page statistics: show the total and average time spent on querying the synces files database
  - ability to decide from which destination a file will be served (if multiple destinations for a file are available) based on user properties (user role, language, location) or whatever other property

## 12.2 Drupal core patch

I had the chance to speak to Andrew “drewish” Morton at DrupalCon DC about this. He is the one who managed to get his proposed Drupal File API patches committed to the current development version of Drupal (which will become Drupal 7). So he definitely is the person to go to for all things concerning files in Drupal right now. I explained to him the need for a unified file URL generation/alteration mechanism and he immediately understood and agreed.

My patch will be against Drupal 6, but the file URL generation mechanism is identical in Drupal 7. So, my patch should be easy to port to Drupal 7.

Drupal already has one function to generate file URLs: `file_create_url($path)`. Unfortunately, this function is only designed to work for files that have been uploaded by users or are generated by modules (e.g. transformations of images). And now the bad news: there is no function through which the URLs for the other files (the ones that aren’t uploaded but are shipped with Drupal core and modules and themes) are generated. To be honest, the current method for generating these URLs is very ugly, although very simple: prepend the base path to the relative file path. So if you want to serve the file `misc/jquery.js` (which is part of Drupal core), then you would write the following code to generate an URL for it:

```
$url = base_path() . 'misc/jquery.js';
```

Andrew and I agreed that since eventually both kinds of files are typically served from the same server(s), it only makes sense to generate their URLs through one function. So the sensible thing to do was to also route the non-uploaded files through the `file_create_url()` function to generate their URLs. And then there would be a function that a module could implement, `custom_file_url_rewrite($path)` which would then allow file URLs to be altered.

So, I wrote a Drupal core patch exactly according to these specifications, and it works great. However, we must fall back to the old mechanisms in case the `custom_file_url_rewrite()` function returned `FALSE` (meaning that the CDN can’t or shouldn’t serve the file). But since there is a distinction between uploaded/generated files and shipping files, we must first determine which kind of file it is. This can be done by looking at the path that was given to `file_create_url()`: if it begins with the path of the directory that the Drupal administrator chose to use for uploaded and generated files, then it is an uploaded/generated file. After this distinction has been made, the original procedures are applied and we’re done.

## 12.3 Implementation

- A simple configuration UI was created using the Forms API [45]. Advanced mode cannot be started if the daemon isn’t configured properly yet (by ensuring the synced files database exists).

- The per-page statistics are rendered through Drupal's `hook_exit()`, which is called just before the end of each page request. It is therefore able to render after the rest of the page is rendered, which of course implies that all file URLs have been created, so it's safe to calculate the statistics.
- A `hook_requirements()` implementation was created, which allows me to add information about the CDN integration module to Drupal's status report page.
- The aforementioned `custom_file_url_rewrite()` function was implemented, which rewrites the URL based on the mode. In basic mode, the CDN URL is automatically inserted into file URLs and in advanced mode, the syncs files database is queried. This is an SQLite database, which the Drupal 6 database abstraction layer does not support. Drupal 7's database abstraction layer does support SQLite, but is still in development (and will be for at least 6 more months). Fortunately, there's also PDO [89], which makes this sufficiently easy.

That's all there is to tell about this module. It's very simple: all complexity is now embedded in the daemon. As it should be.

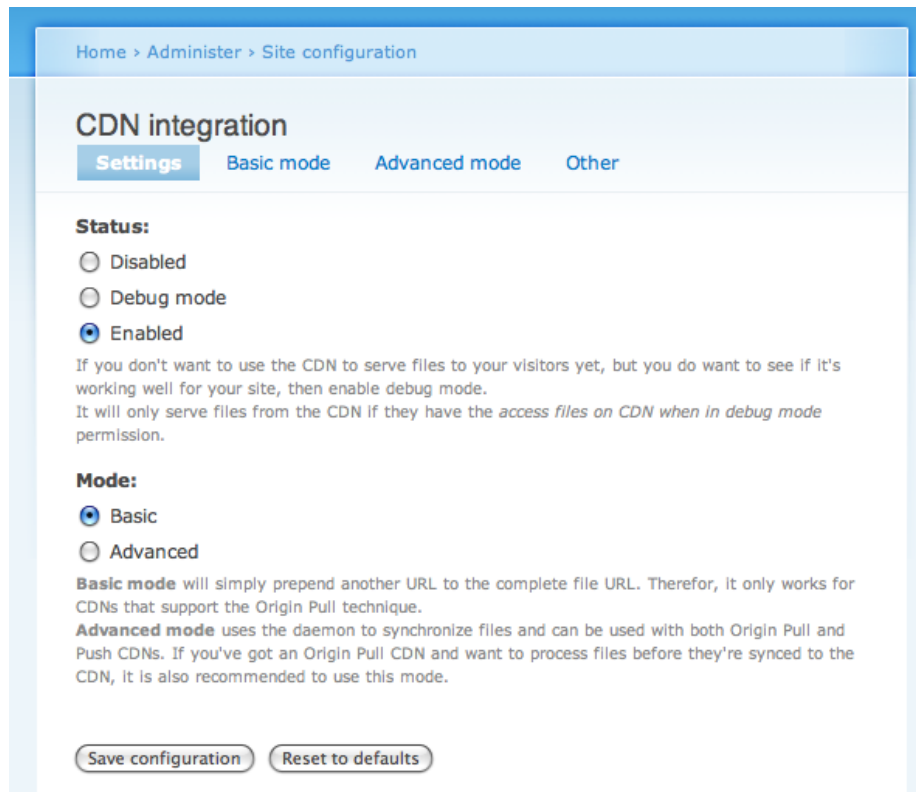
## 12.4 Comparison with the old CDN integration module

In January 2008, I wrote the initial version of the CDN integration module. It was for Drupal 5 instead of Drupal 6 though and it didn't support Origin Pull CDNs, instead, it only supported push CDNs that were accessible over FTP. The synchronization happened from within Drupal, on each cron run. Which means it relied on manual file system scanning (i.e. polling) to detect changes and was prevented by design to perform concurrent syncs, since PHP cannot do that. To top it off, it didn't store anything in the database, but in a serialized array, which had to be unserialized on every page to retrieve the URLs. It should be obvious that this was significantly slower and absolutely unscalable and definitely unusable on any *real* web sites out there.

It had its algorithms right though. You could consider it a very faint preview of what the end result looks like right now.

## 12.5 Screenshots

### The configuration UI



Home > Administer > Site configuration

### CDN integration

**Settings** Basic mode Advanced mode Other

**Status:**

Disabled

Debug mode

Enabled

If you don't want to use the CDN to serve files to your visitors yet, but you do want to see if it's working well for your site, then enable debug mode. It will only serve files from the CDN if they have the *access files on CDN when in debug mode* permission.

**Mode:**

Basic

Advanced

**Basic mode** will simply prepend another URL to the complete file URL. Therefore, it only works for CDNs that support the Origin Pull technique.

**Advanced mode** uses the daemon to synchronize files and can be used with both Origin Pull and Push CDNs. If you've got an Origin Pull CDN and want to process files before they're synced to the CDN, it is also recommended to use this mode.

[Save configuration](#) [Reset to defaults](#)

Figure 17: CDN integration module settings form.

Home > Administer > Site configuration > CDN integration

## CDN integration

Settings **Basic mode** Advanced mode Other

**CDN URL:**

The CDN URL prefix that should be used. Only works for CDNs that support Origin Pull.  
**WARNING:** do not use subdirectories when you're serving CSS files from the CDN. The references to images and fonts from within the CSS files will break because the URLs are no longer valid.

**Allowed extensions:**

Only files with these extensions will be synced.

Figure 18: CDN integration module basic mode settings form.

Home > Administer > Site configuration > CDN integration

## CDN integration

Settings Basic mode **Advanced mode** Other

The synced files database was found and can be opened for reading.

**Synced files database:**

Enter the full path to the daemon's synced files database file.

Figure 19: CDN integration module advanced mode settings form.



Figure 20: CDN integration module other settings form.

## The status report

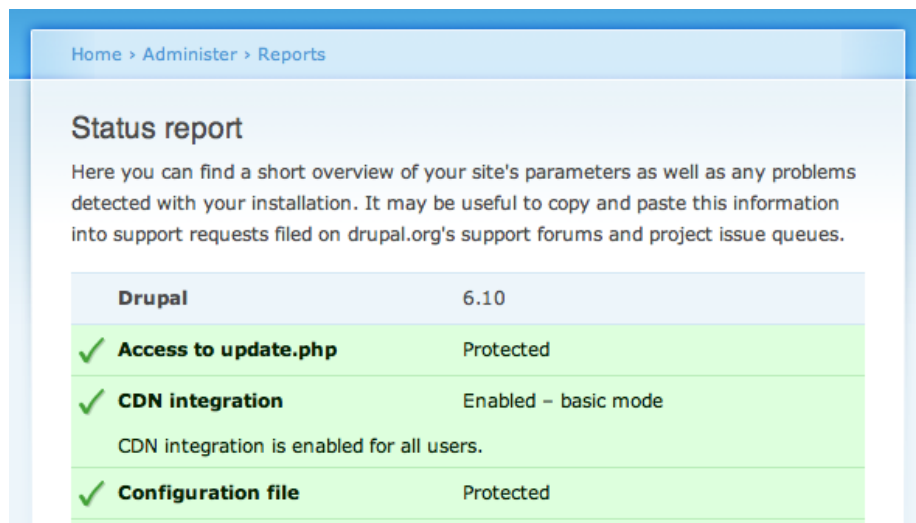


Figure 21: Status report (basic mode, enabled).

Home > Administer > Reports

## Status report

Here you can find a short overview of your site's parameters as well as any problems detected with your installation. It may be useful to copy and paste this information into support requests filed on drupal.org's support forums and project issue queues.

<b>Drupal</b>	6.10
✓ <b>Access to update.php</b>	Protected
⚠ <b>CDN integration</b>	In debug mode – basic mode

CDN integration is only enabled for users with the *access files on CDN when in debug mode* permission

Figure 22: Status report (basic mode, debug mode).

Home > Administer > Reports

## Status report

Here you can find a short overview of your site's parameters as well as any problems detected with your installation. It may be useful to copy and paste this information into support requests filed on drupal.org's support forums and project issue queues.

<b>Drupal</b>	6.10
✓ <b>Access to update.php</b>	Protected
⚠ <b>CDN integration</b>	Disabled

CDN integration is disabled for all users.

Figure 23: Status report (disabled).



Home > Administer > Reports

## Status report

Here you can find a short overview of your site's parameters as well as any problems detected with your installation. It may be useful to copy and paste this information into support requests filed on drupal.org's support forums and project issue queues.

<b>Drupal</b>	6.10
✓ <b>Access to update.php</b>	Protected
✗ <b>CDN integration</b>	Enabled – advanced mode

CDN integration is enabled for all users.

- The synced files database exists.
- The synced files database is readable.
- 744 files have been synced.
- **The daemon is currently not running.**
- 0 files are waiting to be synced.
- 1 files are currently being synced.

Figure 24: Status report (advanced mode, enabled, daemon not running).

Home > Administer > Reports

## Status report

Here you can find a short overview of your site's parameters as well as any problems detected with your installation. It may be useful to copy and paste this information into support requests filed on drupal.org's support forums and project issue queues.

<b>Drupal</b>	6.10
✓ <b>Access to update.php</b>	Protected
✓ <b>CDN integration</b>	Enabled – advanced mode

CDN integration is enabled for all users.

- The synced files database exists.
- The synced files database is readable.
- 744 files have been synced.
- The daemon is currently running.
- 0 files are waiting to be synced.
- 1 files are currently being synced.

Figure 25: Status report (advanced mode, enabled, daemon running).

## The per-page statistics

### CDN integration statistics for *admin/reports/status*

- Total number of files on this page: **9**.
- Number of files available on CDNs: **6** (67% coverage).
- Number of files served from the server *origin pull cdn*: 4
- Number of files served from the server *ftp push cdn*: 2
- Total time it took to look up the CDN URLs for these files: 4.339 ms, or 0.482 ms on average per file.
- The files that are not (yet?) synchronized to the CDN:
  - [sites/default/files/css/08a34b90f4b5cc9f2c4fc8105a769a2a.css](#)
  - [sites/default/files/css/55a34626dc4c012013e4107c0216bbec.css](#)
  - [sites/default/files/js/df1aaa97d81ce59ca05c5b53e4e0fbe3.js](#)
- The files that are synchronized to the CDN:
  - [themes/garland/logo.png](#) (server: origin pull cdn)
  - [misc/favicon.ico](#) (server: ftp push cdn)
  - [misc/powered-blue-80x15.png](#) (server: ftp push cdn)
  - [modules/dblog/dblog.css](#) (server: origin pull cdn)
  - [sites/all/modules/admin\\_menu/admin\\_menu.css](#) (server: origin pull cdn)

Figure 26: Per-page statistics.

## 13 Used technologies

- Languages
  - PHP
  - JavaScript
  - Python
  - SQL
- Frameworks
  - Drupal (Forms API, Batch API, menu system, Schema API, etc.)
  - jQuery
  - Episodes [40]
  - Django's [71] `Storage` class [72] and its dependencies
- APIs/libraries
  - Browser.php [48]
  - Google Chart API [47]
  - FSEvents [51, 52] (through the Python-Objective-C bridge [53])
  - inotify [49] (through the Python pyinotify [50] module)
  - SQLite [54] (through the Python sqlite3 [57] module and the PHP PDO [89] database abstraction layer)
  - django-storages [73]
  - cssutils [67]
- Uses the following 3rd party applications
  - ImageMagick [63]
  - pngcrush [64]
  - jpegtran [65]
  - gifsicle [66]
  - YUI Compressor [68]
- Supports the following storage systems
  - FTP (via django-storages, through the Python ftplib [77] module)
  - Amazon S3 [78] (via django-storages, through the Python boto [80] module)
  - Amazon CloudFront [79] (via django-storages, through the Python boto [80] module)
- Integrates with the following applications
  - Apache HTTP Server

## 14 Feedback from businesses

TODO

## 15 Conclusion

TODO

## References

- [1] *Design Fast Websites*, Nicole Sullivan, 2008, <http://www.slideshare.net/stubbornella/designing-fast-websites-presentation>
- [2] *We're all guinea pigs in Google's search experiment*, Stephen Shankland, [http://news.cnet.com/8301-10784\\_3-9954972-7.html](http://news.cnet.com/8301-10784_3-9954972-7.html)
- [3] *High Performance Web Sites*, Steve Souders, 2007, O'Reilly, <http://stevesouders.com/hpws/>
- [4] *Usage statistics for Drupal*, <http://drupal.org/project/usage/drupal>
- [5] *Improving Drupal's page loading performance*, Wim Leers, January 2008, <http://wimleers.com/article/improving-drupals-page-loading-performance>
- [6] *Content Owners Struggling To Compare One CDN To Another*, March 2008, [http://blog.streamingmedia.com/the\\_business\\_of\\_online\\_vi/2008/03/content-owners.html](http://blog.streamingmedia.com/the_business_of_online_vi/2008/03/content-owners.html)
- [7] *How Is CDNs Network Performance For Streaming Measured?*, August 2007, [http://blog.streamingmedia.com/the\\_business\\_of\\_online\\_vi/2007/08/cdns-network-pe.html](http://blog.streamingmedia.com/the_business_of_online_vi/2007/08/cdns-network-pe.html)
- [8] *UA Profiler*, Steve Souders, 2008, <http://stevesouders.com/ua/>
- [9] *Cuzillion*, Steve Souders, 2008, <http://stevesouders.com/cuzillion/>
- [10] *Cuzillion*, Steve Souders, 2008, <http://www.stevesouders.com/blog/2008/04/25/cuzillion/>
- [11] *Hammerhead*, Steve Souders, 2008, <http://stevesouders.com/hammerhead/>
- [12] *Hammerhead: moving performance testing upstream*, Steve Souders, September 2008, <http://www.stevesouders.com/blog/2008/09/30/hammerhead-moving-performance-testing-upstream/>
- [13] *Firebug*, <http://getfirebug.com/>
- [14] *Fasterfox*, <http://fasterfox.mozdev.org/>
- [15] *YSlow*, Steve Souders, 2007, <http://developer.yahoo.com/yslow/>
- [16] *Exceptional Performance*, 2007, <http://developer.yahoo.com/performance/index.html>
- [17] *Best Practices for Speeding Up Your Web Site*, 2008, <http://developer.yahoo.com/performance/rules.html>
- [18] *YSlow: Yahoo's Problems Are Not Your Problems*, Jeff Atwood, 2007, <http://www.codinghorror.com/blog/archives/000932.html>
- [19] *YSlow 2.0 early preview in China*, Yahoo! Developer Network, 2008, [http://developer.yahoo.net/blog/archives/2008/12/yslow\\_20.html](http://developer.yahoo.net/blog/archives/2008/12/yslow_20.html)

- [20] *State of Performance 2008*, Steve Souders, 2008, <http://www.stevesouders.com/blog/2008/12/17/state-of-performance-2008/>
- [21] *Apache JMeter*, <http://jakarta.apache.org/jmeter/>
- [22] *Load test your Drupal application scalability with Apache JMeter*, John Quinn, 2008, <http://www.johnandcailin.com/blog/john/load-test-your-drupal-application-scalability-apache-jmeter>
- [23] *Load test your Drupal application scalability with Apache JMeter: part two*, John Quinn, 2008, <http://www.johnandcailin.com/blog/john/load-test-your-drupal-application-scalability-apache-jmeter:-part-two>
- [24] *Gomez*, <http://www.gomez.com/>
- [25] *Keynote*, <http://www.keynote.com/>
- [26] *WebMetrics*, <http://www.webmetrics.com/>
- [27] *Pingdom*, <http://pingdom.com/>
- [28] *AJAX*, <http://en.wikipedia.org/wiki/AJAX>
- [29] *Selenium*, <http://seleniumhq.org/>
- [30] *Keynote KITE*, <http://kite.keynote.com/>
- [31] *Gomez Script Recorder*, [http://www.gomeznetworks.com/help/Gomezu/main/Gomez\\_university/3\\_Gomez\\_Script\\_Recorder/toc.htm](http://www.gomeznetworks.com/help/Gomezu/main/Gomez_university/3_Gomez_Script_Recorder/toc.htm)
- [32] *WhitePages*, <http://whitepages.com/>
- [33] *Velocity 2008, Jiffy: Open Source Performance Measurement and Instrumentation*, Scott Ruthfield, 2008, <http://en.oreilly.com/velocity2008/public/schedule/detail/4404>
- [34] *Velocity 2008, video of the Jiffy presentation*, Scott Ruthfield, 2008, <http://blip.tv/file/1018527>
- [35] *Jiffy*, <http://code.google.com/p/jiffy-web/>
- [36] *Jiffy Firebug Extension*, <http://billwscott.com/jiffyext/>
- [37] *Episodes: a Framework for Measuring Web Page Load Times*, Steve Souders, July 2008, <http://stevesouders.com/episodes/paper.php>
- [38] *Episodes: a shared approach for timing web pages*, Steve Souders, 2008, <http://stevesouders.com/docs/episodes-tae-20080930.ppt>
- [39] *Google Analytics*, <http://google.com/analytics>
- [40] *Episodes*, Steve Souders, 2008, <http://stevesouders.com/episodes/>
- [41] *Episodes: a Framework for Measuring Web Page Load Times*, Steve Souders, July 2008, <http://stevesouders.com/episodes/paper.php>

- [42] *Episodes Drupal module*, Wim Leers, 2009, <http://drupal.org/project/episodes>
- [43] *Episodes Example*, Steve Souders, 2008, <http://stevesouders.com/episodes/example.php>
- [44] *Batch API*, Drupal 6, <http://api.drupal.org/api/group/batch/6>
- [45] *Forms API*, Drupal 6, [http://api.drupal.org/api/group/form\\_api/6](http://api.drupal.org/api/group/form_api/6)
- [46] *Hierarchical Select module*, Wim Leers, [http://drupal.org/project/hierarchical\\_select](http://drupal.org/project/hierarchical_select)
- [47] *Google Chart API*, <http://code.google.com/apis/chart/>
- [48] *Browser.php*, Chris Schuld, 2009, <http://chrisschuld.com/projects/browser-php-detecting-a-users-browser-from-php/>
- [49] *inotify*, <http://en.wikipedia.org/wiki/Inotify>
- [50] *pyinotify*, <http://pyinotify.sourceforge.net/>
- [51] *FSEvents Programming Guide*, 2008, [http://developer.apple.com/documentation/Darwin/Conceptual/FSEvents\\_ProgGuide/Introduction/Introduction.html](http://developer.apple.com/documentation/Darwin/Conceptual/FSEvents_ProgGuide/Introduction/Introduction.html)
- [52] *FSEvents review*, John Siracusa, 2007, <http://arstechnica.com/apple/reviews/2007/10/mac-os-x-10-5.ars/7>
- [53] *PyObjC*, <http://pyobjc.sourceforge.net/>
- [54] *SQLite*, <http://www.sqlite.org/>
- [55] *How SQLite is Tested*, <http://www.sqlite.org/testing.html>
- [56] *shelve - Python object persistence*, <http://docs.python.org/library/shelve.html>
- [57] *pysqlite*, <http://docs.python.org/library/sqlite3.html>
- [58] *smush.it*, <http://smush.it/>
- [59] *Image Optimization Part 1: The Importance of Images*, Stoyan Stefanov, 2008, <http://yuiblog.com/blog/2008/10/29/imageopt-1/>
- [60] *Image Optimization Part 2: Selecting the Right File Format*, Stoyan Stefanov, 2008, <http://yuiblog.com/blog/2008/11/04/imageopt-2/>
- [61] *Image Optimization, Part 3: Four Steps to File Size Reduction*, Stoyan Stefanov, 2008, <http://yuiblog.com/blog/2008/11/14/imageopt-3/>
- [62] *Image Optimization, Part 4: Progressive JPEG ... Hot or Not?*, Stoyan Stefanov, 2008, <http://yuiblog.com/blog/2008/12/05/imageopt-4/>
- [63] *ImageMagick*, <http://imagemagick.org/>
- [64] *pngcrush*, <http://pmt.sourceforge.net/pngcrush/>



- [65] *jpegtran*, <http://jpegclub.org/>
- [66] *gifsicle*, <http://www.lcdf.org/gifsicle/>
- [67] *cssutils*, <http://cthedot.de/cssutils/>
- [68] *YUI Compressor*, <http://www.julienlecomte.net/blog/2007/08/11/>
- [69] *Rhino*, <http://www.mozilla.org/rhino/>
- [70] *JSMIn, The JavaScript Minifier*, Douglas Crockford, 2003, <http://javascript.crockford.com/jsmin.html>
- [71] *Django*, <http://www.djangoproject.com/>
- [72] *Writing a custom storage system*, Django 1.0 documentation, <http://docs.djangoproject.com/en/1.0/howto/custom-file-storage/>
- [73] *django-storages*, David Larlet et al., <http://code.welldev.org/django-storages/wiki/Home>
- [74] *FTPStorage: saving large files + more robust exists()*, Wim Leers, 2009, <http://code.welldev.org/django-storages/issue/4/ftpstorage-saving-large-files-+-more-robust>
- [75] *S3BotoStorage: set Content-Type header, ACL fixed, use HTTP and disable query auth by default*, Wim Leers, 2009, <http://code.welldev.org/django-storages/issue/5/s3botostorage-set-content-type-header-acl-fixed-use-http-and-disable-query-auth-by>
- [76] *SymlinkOrCopyStorage: new custom storage system*, Wim Leers, 2009, <http://code.welldev.org/django-storages/issue/6/symlinkorcopystorage-new-custom-storage>
- [77] *ftplib — FTP protocol client*, <http://docs.python.org/library/ftplib.html>
- [78] *Amazon S3*, <http://aws.amazon.com/s3/>
- [79] *Amazon CloudFront*, <http://aws.amazon.com/cloudfront/>
- [80] *boto*, <http://code.google.com/p/boto/>
- [81] *MogileFS*, <http://www.danga.com/mogilefs/>
- [82] *Apache CouchDB*, <http://couchdb.apache.org/>
- [83] *Stopping and Restarting - Apache HTTP Server*, <http://httpd.apache.org/docs/2.2/stopping.html>
- [84] *Pipes and Filters*, [http://en.wikipedia.org/wiki/Pipes\\_and\\_filters](http://en.wikipedia.org/wiki/Pipes_and_filters)
- [85] *Pipes and Filters*, Jorge Luis Ortega Arjona, Department of Computer Science of the University College London, <http://www.cs.ucl.ac.uk/staff/J.Ortega-Arjona/patterns/PF.html>

- [86] *Pipe-and-filter*, Jike Chong; Arlo Faria; Satish Nadathur; Youngmin Yi, Electrical Engineering and Computer Sciences department of UC Berkely, <http://parlab.eecs.berkeley.edu/wiki/patterns/pipe-and-filter>
- [87] *Pipes and Filters*, Enterprise Integration Patterns, <http://www.eaipatterns.com/PipesAndFilters.html>
- [88] *Currying*, <http://en.wikipedia.org/wiki/Currying>
- [89] *PDO*, <http://php.net/pdo>