

Improving Drupal's page loading performance

Wim Leers

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

Promotor: Prof. dr. Wim Lamotte
Co-promotor: dr. Peter Quax
Mentors: Stijn Agten & Maarten Wijnants

Hasselt University
Academic year 2008-2009

Abstract

This bachelor thesis is about improving Drupal's page loading performance through integrating Drupal with a CDN. Page loading performance is about reducing the time it takes to load a web page. Because reducing that time also reduces the time required to access information, increases the number of satisfied visitors, and if the web site is commercial, it increases revenue.

Before you can prove that improvements are made, you need a tool to measure that. So first, a comparison is made of available page loading performance profiling tools (and related tools). Episodes is chosen because it is the only tool that measures the real-world page loading performance. This requires tight integration with Drupal though, so a module was written to integrate Episodes with Drupal. A companion module to visualize the collected measurements through basic charts was also written.

Next, a daemon was written to synchronize files to a CDN (actually, any kind of file server). It was attempted to make the configuration as self-explanatory as possible. The daemon is capable of processing the file before it is synced, for example to optimize images or compress CSS and JavaScript files. A variety of transporters (for different protocols) is available to transport the file to file servers. According to the configuration file, files are detected through the operating system's file system monitor and then processed and transported to their destination servers. The resulting URLs at which the files are available are stored in a database.

Then, a Drupal module was written that makes it easy to integrate Drupal with a CDN (both with and without the daemon). A patch for Drupal core had to be written to make it possible to alter the URLs to static files (CSS, JavaScript, images, and so on). To make this functionality part of the next version of Drupal core, a patch for that version was also submitted.

Finally, a test case was built. A high-traffic web site with a geographically dispersed audience was migrated to Drupal and the Episodes integration was enabled. The first period, no CDN integration was enabled. Then the daemon was installed and CDN integration was enabled. Files were being synced to a static file server in Belgium and a North-American CDN, and visitors were assigned to either one, based on their geographical location. This second period, with CDN integration enabled, was also measured using Episodes and conclusions were drawn from this.

Preface

When I wrote a custom proposal for a bachelor thesis, it was quickly approved by my promotor, Prof. dr.Wim Lamotte. I would like to thank him for making this bachelor thesis possible. He also approved of writing this thesis in the open, to get feedback from the community and to release all my work under the GPL, which will hopefully ensure it will be used. I have worked to the best of my abilities to try to ensure that my work can be used in real world applications.

During the creation of this bachelor thesis, I have often received very useful feedback from my mentors, Maarten Wijnants and Stijn Agten. My sincere thanks go to them. I would also like to thank dr. Peter Quax, co-promotor of this bachelor thesis.

I would also like to thank Rambla and SimpleCDN for providing free CDN accounts for testing and SlideME for providing feedback on my work.

Finally, I would like to thank my parents and my brother, whose support has been invaluable.

Dutch summary/Nederlandstalige samenvatting

Het doel van deze bachelorproef is het verbeteren van de “page loading performance” van Drupal.

Drupal is een systeem om websites mee te bouwen en is bedoeld voor zowel ontwikkelaars als eindgebruikers: het voorziet zowel uitgebreide API's als een rijk ecosysteem aan kant-en-klare modules, die kunnen gedownload en geïnstalleerd worden in minder dan een minuut. Het is geschreven in PHP, omdat die taal op de meeste servers beschikbaar is en één van de doelstellingen van Drupal is om op zoveel mogelijk servers te werken. Zo is het aantal potentiële gebruikers het grootst, want het maakt het gebruik van Drupal goedkoper. Drupal is er ook op gericht om zoveel mogelijk te innoveren en om de laatste trends te volgen — of er op vooruit te lopen. Het is een volwassen open source software project dat wordt gebruikt door vele bekende instanties, waaronder de Belgische overheid, Disney, de NASA, Harvard university en de Verenigde Naties. Honderdduizenden websites gebruiken Drupal. Het verbeteren van de page loading performance van Drupal kan dus een effect hebben op een groot aantal websites. Een van de meest effectieve methodes om de page loading performance te verbeteren, is het gebruik van een CDN.

Een CDN is een verzameling van webservern die verspreid staan over meerdere locaties om gegevens efficiënter af te leveren bij gebruikers. De server die geselecteerd wordt om gegevens aan een specifieke gebruiker te leveren wordt meestal gedaan op basis van de afstand in het netwerk, waarbij dichterbij beter is. Er zijn twee soorten CDN wat betreft het plaatsen van bestanden op de CDN: push en pull. Pull vereist vrijwel geen werk: URLs moeten aangepast worden, waarbij de domeinnaam van de CDN geplaatst wordt waar voorheen de domeinnaam van de website stond. De CDN downloadt dan vanzelf de bestanden die het moet verzenden aan de gebruiker van de server van de website. Dit wordt de *Origin Pull* techniek genoemd. Anderzijds zijn er ook CDNs die een push-mechanisme ondersteunen, waarbij door middel van bijvoorbeeld FTP de bestanden op de CDN kunnen geplaatst worden. De CDN downloadt dus niet vanzelf de bestanden, die moeten er door de eigenaar van de website op geplaatst worden.

Mijn doelstelling was om drie soorten CDN's te ondersteunen:

1. iedere CDN die Origin Pull ondersteunt
2. iedere CDN die FTP ondersteunt
3. Amazon S3/CloudFront, dit is een specifieke CDN met een eigen protocol. Omdat deze zo populair is, heb ik er voor gekozen om deze ook expliciet te ondersteunen.

Dit lijken de de drie meest gebruikte soorten CDN's te zijn.

Page loading performance gaat over het minimaliseren van de tijd die nodig is om een webpagina in te laden. Want snellere websites betekent tevredenere bezoekers die vaker terugkomen en, indien je website commercieel is, meer omzet.

Zo wezen bijvoorbeeld tests van Google uit dat een halve seconde extra tijd om zoekresultaten te laden, een vermindering van twintig procent van het aantal zoekacties met zich meebracht. Amazon merkte dat iedere honderd milliseconde aan extra tijd om een webpagina in te laden in een daling van één procent van het aantal verkopen resulteerde. Als je dan nog weet dat het helemaal niet uitzonderlijk is om laadtijden van vijf seconden en meer te hebben, wordt al snel duidelijk dat de impact groot kan zijn.

Om te garanderen dat mijn pogingen om de page loading performance van Drupal te verbeteren, was het echter noodzakelijk om de resultaten te kunnen meten. Na het analyseren van een brede waaier aan page loading performance profiling tools (en gerelateerde tools), werd het al snel duidelijk dat Episodes (dit was slechts een prototype, geschreven door Steve Souders — dit is de persoon die het fenomeen page loading performance en het nut van het optimaliseren ervan, bekend heeft gemaakt) was de beste kandidaat: het is de enige tool die de mogelijkheid heeft om de wérkelijke page loading performance te meten, omdat het metingen doet in de browser van iedere bezoeker van de website, bij iedere paginaweergave. De gemeten episodes worden door middel van een GET request en een uitgebreide query string naar een Apache server gelogd naar een Apache log. Deze tool heeft bovendien ook het potentieel om dé standaard te worden in de loop van de komende jaren. Het is zelfs niet ondenkbaar dat het ingebouwd zal worden in toekomstige browsers.

Om het te gebruiken in mijn bachelorproef, heb ik de code opgekuist en het klaar gemaakt (of tenminste bruikbaar) voor gebruik op een Drupal website, via een nieuwe Drupal module: de Episodes module. Deze integratie gebeurt op zo'n manier dat alle “Drupal behaviors” (dit zijn alle JavaScript “behaviors” (gedragingen) die gedefinieerd worden door middel van een vastgelegde Drupal JavaScript API) automatisch worden gemeten. Alles dat de eigenaar van de website moet doen, is enkele veranderingen in zijn Drupal “theme” (letterlijk: thema, het design van de website dus) aan te brengen om er voor te zorgen dat al dat wat kan gemeten worden, ook effectief gemeten wordt.

Deze module is klaar voor gebruik in productie.

Ik heb tevens een bijbehorende Episodes Server module gemaakt. Via deze module is het mogelijk om de logs die verzameld zijn door middel van Episodes te importeren en de metingen te visualiseren door middel van grafieken (gegenereerd aan de hand van Google Chart API). Dankzij deze grafieken kan je de werkelijke page loading performance over een tijdspanne evalueren. Het is zelfs mogelijk om de page loading performance van meerdere landen tegelijk te vergelijken met de globale page loading performance. Het laat je ook toe om te zien welke episodes momenteel het langst duren en dus het best geschikt zijn voor optimalisatie.

Deze module is nog niet klaar voor gebruik in productie, maar het is een goede basis om van te beginnen. De code die de logs in de database importeert, werkt gegarandeerd dankzij unit tests.

Dan is er natuurlijk de daemon om bestanden te synchroniseren. Dit was het belangrijkste deel van deze bachelorproef. Hoe raar het ook mag lijken, er lijkt niets vergelijkbaar te bestaan, of tenminste niet publiekelijk beschikbaar (zelfs geen commerciële programma's). Als het zou bestaan, zou ik het zeker al vernomen hebben van een van de tientallen mensen die op de hoogte zijn van het

concept en doel van mijn bachelorproef.

Ik ben begonnen met het ontwerp van het configuratiebestand. De configuratie van de daemon gebeurt door middel van een XML bestand dat is ontworpen om gemakkelijk te zijn in gebruik, op voorwaarde dat je bekend bent met de terminologie. Verscheidene mensen die zich thuis voelen in de terminologie werden gevraagd om een voorbeeld configuratiebestand te bekijken en ze antwoordden meteen dat het logisch in elkaar zat. Het is belangrijk dat dit gemakkelijk is, want het is de interface naar de daemon toe.

De daemon werd gesplitst in grote componenten en ik ben begonnen met de eenvoudigste, omdat ik dit zou schrijven in Python, een taal die ik nooit eerder had gebruikt. Ik heb voor deze taal gekozen omdat volgens de geruchten het je leven als programmeur een stuk makkelijker zou maken, deels dankzij de beschikbaarheid van modules voor vrijwel alles dat je je kan inbeelden. Gelukkig bleek dat grotendeels waar te zijn, hoewel ik wel lange tijd gevreesd heb dat ik alle code voor het transporteren van bestanden (transporters) zelf zou moeten gaan schrijven. Dat zou een vrijwel onmogelijke opdracht geweest zijn, gegeven de hoeveelheid tijd.

Dit is dan ook de reden waarom de daemon eigenlijk eenvoudigweg een verzameling Python modules is. Deze modules kunnen in eender welke applicatie gebruikt worden, bijvoorbeeld de `fsmonitory.py` module — die bestandssysteem monitors op verschillende besturingssystemen abstraheert en zo een cross-platform API creëert — kan makkelijk hergebruikt worden in andere applicaties. Dus heb ik een relatief grote verzameling Python modules geschreven: `config.py`, `daemon_thread_runner.py`, `filter.py`, `fsmonitor.py` (met een subclass voor ieder ondersteund besturingssysteem), `pathscanner.py`, `persistent_list.py`, `persistent_queue.py`, `processor.py` (met een verzameling subclasses, één voor iedere processor) en `transporter.py` (met subclasses die zeer dunne wrappers rond Django custom storage systems zijn). Telkens indien het haalbaar was, heb ik unit tests geschreven. Maar omdat er in deze applicatie veel bestand I/O en netwerk I/O aan te pas komt, was dit vaak extreem complex om te doen en dus overgeslagen, ook omdat de hoeveelheid beschikbare tijd beperkt was. Voor `fsmonitor` ondersteuning in Linux kon ik verder bouwen op de `pyinotify` module en voor de transporters zag ik de mogelijkheid om Django's custom storage systems geheel te hergebruiken, waardoor ik zonder al te veel moeite ondersteuning heb voor FTP, Amazon S3 en “Symlink or Copy” (een speciaal custom storage system, om verwerkte bestanden ook te kunnen synchroniseren met Origin Pull CDNs). Django is een framework om websites mee te bouwen (in tegenstelling tot Drupal is het enkel geschikt voor developers) en daarvan heb ik dus één API (en zijn dependencies) hergebruikt. Het gevolg is dus dat veranderingen die gemaakt worden aan de transporters in de daemon, weer kunnen worden teruggegeven en vice versa. Ik heb enkele bugfixes doorgegeven die reeds goed bevonden zijn en nu dus deel uitmaken van die code.

Dit heeft een interessant neveneffect: `arbitrator.py`, de module die al deze op zich zelf staande modules samenbindt tot één geheel (die dus arbitreert tussen de verscheidene modules), kan eenvoudig compleet gerefactored worden. Hoewel het bijna duizend regels code is (maar veel regels daarvan zijn commentaar), kan men eenvoudig de hele arbitrator herschrijven, omdat het enkel logica bevat die de losse modules aan elkaar linkt. Dus indien er bijvoorbeeld een bottleneck zou gevonden worden die zich enkel in bepaalde situaties voordoet omwille van een fout in het ontwerp van de arbitrator, kan dit relatief eenvoudig opgevangen

worden, omdat alle logica van de daemon in een enkele module zit. Omdat het onmogelijk is om zeker te zijn dat de daemon correct en betrouwbaar werkt in iedere omgeving en iedere mogelijke configuratie, is het aan te raden dat een bedrijf eerst haar use case simuleert en verifieert dat de daemon zoals gewenst functioneert in die simulatie. Hopelijk trekt dit project voldoende mensen aan die er aan werken om het verder geschikt te maken voor meer situaties.

Een Drupal module om de integratie met CDNs te vereenvoudigen werd ook geschreven: de CDN integratie module. Echter, voordat deze kon geschreven worden, was het nodig om een patch voor Drupal core te schrijven, omdat het nodig is om de URLs naar bestanden te kunnen aanpassen. Indien deze URLs niet aanpasbaar zijn (zoals het geval is voor Drupal 6), kunnen ze ook niet aangepast worden om naar een CDN te verwijzen.

Een patch voor Drupal 7 (deze versie van Drupal is momenteel in ontwikkeling) — met unit tests want dat is een vereiste — om deze functionaliteit deel te laten uitmaken van Drupal in de toekomst, heeft zeer positieve reviews gekregen, maar moet nog steeds door het minutieuze peer review proces gaan. Het is zeer waarschijnlijk dat het binnenkort gecommitt zal worden.

Er zijn twee modi beschikbaar in de Drupal module: eenvoudig en geavanceerd. In de geavanceerde modus kan enkel gebruik gemaakt worden van Origin Pull CDN's. Maar omdat het gebruiken van dit soort CDN's nu zeer eenvoudig wordt dankzij deze module, terwijl het vroeger een reeks manuele stappen vereiste, is dit alleen al erg nuttig. Echter, in de geavanceerde modus wordt het pas echt interessant: dan wordt de database van gesynchroniseerde bestanden gebruikt die door de daemon werd aangemaakt en wordt onderhouden. Dan wordt de URL van een bestand op de CDN opgezocht, waarna deze URL wordt gebruikt. Het is zelfs mogelijk om een speciale callback functie te implementeren die kan gebruikt worden om een specifieke server te selecteren, op basis van de eigenschappen van de gebruiker (locatie, type lidmaatschap of wat dan ook). Deze module is ook klaar voor gebruik in productie.

De feedback van bedrijven was teleurstellend wat betreft de hoeveelheid maar overweldigend positief. Op meer positieve feedback zou ik niet gehoopt kunnen hebben. Het potentieel van de daemon werd sterk gewaardeerd. De codestructuur van de daemon werd beschreven als “duidelijk en zelfverklarend” en de documentatie (van de daemon zelf en de beschrijving ervan in de bachelorproef tekst) als “zeer duidelijk”. Het zorgde er blijkbaar zelfs voor dat een reviewer er spijt van kreeg dat hij zijn bachelor graad niet voltooid heeft. Deze reviewer was zelfs zo enthousiast dat hij al begonnen was met het schrijven van patches voor de daemon, zodat die beter inzetbaar was in zijn infrastructuur. Dit suggereert dat het mogelijk haalbaar is dat de daemon een levendig open source project wordt.

Ten slotte bevestigden de resultaten van mijn test case de stelling dat het integreren van Drupal met een CDN de page loading performance kan verbeteren. Hoewel de resultaten (die gelogd worden door middel van de Episodes module) niet zo expliciet waren als ze zouden kunnen geweest zijn voor een website rijk aan media (mijn test case was een website die arm was aan media), was het verschil nog steeds duidelijk te onderscheiden in de grafieken (die gegenereerd werden door de Episodes Server module). Ondanks het feit dat de website al

geoptimaliseerd was aan de hand van de mechanismen die standaard in Drupal aanwezig zijn, resulteerde de integratie met een CDN (via de CDN integratie module en de daemon) in een duidelijke algemene wereldwijde verbetering van de page loading performance.

Contents

1 Terminology	1
2 Definition	3
3 Drupal	4
4 Why it matters	6
5 Key Properties of a CDN	7
6 Profiling tools	9
6.1 UA Profiler	9
6.2 Cuzillion	9
6.3 YSlow	10
6.4 Hammerhead	12
6.5 Apache JMeter	14
6.6 Gomez/Keynote/WebMetrics/Pingdom	15
6.6.1 Limited number of measurement points	15
6.6.2 No real-world browsers	15
6.6.3 Unsuitable for Web 2.0	16
6.6.4 Paid & closed source	16
6.7 Jiffy/Episodes	16
6.7.1 Jiffy	16
6.7.2 Episodes	17
6.8 Conclusion	20
7 The state of Drupal's page loading performance	21

8	Improving Drupal: Episodes integration	22
8.1	The goal	22
8.2	Making episodes.js reusable	24
8.3	Episodes module: integration with Drupal	25
8.3.1	Implementation	25
8.3.2	Screenshots	27
8.4	Episodes Server module: reports	30
8.4.1	Implementation	30
8.4.2	Screenshots	31
8.4.3	Desired future features	31
8.5	Insights	33
8.6	Feedback from Steve Souders	34
9	Daemon	35
9.1	Goals	35
9.2	Configuration file design	37
9.3	Python modules	38
9.3.1	filter.py	38
9.3.2	pathscanner.py	40
9.3.3	fsmonitor.py	41
9.3.4	persistent_queue.py and persistent_list.py	43
9.3.5	Processors	44
9.3.6	Transporters	48
9.3.7	config.py	52
9.3.8	daemon_thread_runner.py	53
9.4	Putting it all together: arbitrator.py	53

9.4.1	The big picture	53
9.4.2	The flow	54
9.4.3	Pipeline design pattern	56
9.5	Performance tests	59
9.6	Possible further optimizations	60
9.7	Desired future features	60
10	Improving Drupal: CDN integration	61
10.1	Goals	61
10.2	Drupal core patch	62
10.3	Implementation	63
10.4	Comparison with the old CDN integration module	63
10.5	Screenshots	64
11	Used technologies	70
12	Feedback from businesses	71
13	Test case: DriverPacks.net	74
14	Conclusion	84

1 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 defines the root of a web site. E.g. if the site `http://example.com/` is where a web site lives, then the base path is `/`. If you have got another web site at `http://example.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, et cetera.

CSS sprite An image that actually contains a grid of other images. Through CSS, each image in the grid can then be accessed (and displayed to the end user). The benefit is that instead of having as many HTTP requests as there are images in the grid, there is now a single HTTP request, reducing the number of round trips and thereby increasing the perceived page loading speed.

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/example.com`.

Drupal behaviors Behaviors are event-triggered actions that attach to HTML elements, enhancing default non-JavaScript UIs. Through this system, behaviors are also attached automatically to new HTML elements loaded through AHAH/AJAX and HTML elements to which the behaviors have already been applied are automatically skipped.

episode An episode in the page loading sequence.

Episodes The Episodes framework [52] (note the capital 'e').

internationalization The process of designing a software application so that it can be adapted to various languages and regions without engineering change.

lazy loading Deferring the loading of something until it is actually needed. In the context of web pages, lazy loading a file implies that it will not be loaded until the end user will actually get to see it.

localization The process of adapting internationalized software for a specific region or language by adding locale-specific components and translating text.

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.

prefetching Loading something when it not yet needed. In the context of web pages, prefetching a file implies that it will be cached by the browser before it is actually used in a web page.

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.

2 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 is not 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 [1]. The other 80-90% of the time is spent on loading the components (CSS stylesheets, JavaScript, images, movies, et cetera) in the page (front-end only). Figure 1 clarifies this visually:

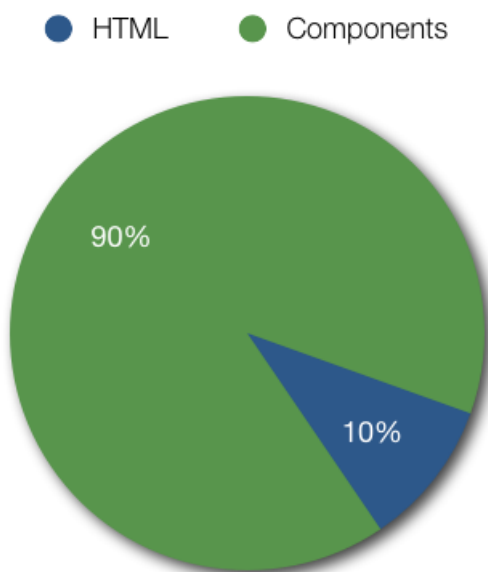


Figure 1: End user response time of a typical web page.

It should be obvious now that it is far more effective to focus on front-end performance than it is to focus on back-end performance, because it has got a greater potential. It is also easier to optimize than the back-end, because instead of having to profile the entire codebase through which the page is generated (which is necessary for optimizing the back-end performance), you can simply change where in the HTML files are being referenced and possibly also replacing the URLs to use a CDN instead. These measures are clearly far more easy to implement.

3 Drupal

Drupal [2] is a content management system (CMS), although it has become more of a content management framework (CMF). The difference between the two is that the former is a system with predefined rules, or with relatively little flexibility. The latter is — as the name already indicates — a framework which still needs to be configured to suit your needs and therefore offers more flexibility.

History

It is an open source project, started in 2000 by Dries Buytaert, whom was then still studying at the University of Antwerp. He built a small news web site with a built-in web board, allowing his friends in the same dorm to leave notes or to announce when they were having dinner. After graduation, they decided they wanted to stay in touch with each other, so they wanted to keep this site online. Dries wanted to register the domain name dorp.org (the Dutch word for “village”), which was considered a fitting name. But he made a typo and registered drop.org.

drop.org’s audience changed as its members began talking about new web technologies, such as syndication, rating and distributed authentication. The ideas resulting from those discussions were implemented on drop.org itself.

Only later, in 2001, Dries released the software behind drop.org as “Drupal”. The purpose was to enable others to use and extend the experimentation platform so that more people could explore new paths for development. The name Drupal, pronounced “droo-puhl,” derives from the English pronunciation of the Dutch word “druppel,” which means “drop” .



Figure 2: Drupal’s mascotte: Druplicon.

What makes it different?

There are a couple of things that separate Drupal from most other CMSes and CMFs. For starters, Drupal has a set of principles [4] it strictly adheres to, amongst which is this one:

Drupal should also have minimal, widely-available server-side software requirements. Specifically, Drupal should be fully operational on a platform with a web server, PHP, and either MySQL or PostgreSQL.

This is the reason PHP was chosen as the language to write Drupal in. PHP is the justification for some people to not even try Drupal. But it is also a reason why so many web sites today are running Drupal, and why its statistics (and the popularity of its web site) have been growing exponentially for years [5, 6]. By settling for the lowest common denominator and creating a robust, flexible platform on top of that, it can scale from a simple blogger (such as myself) to the huge media company (such as Sony BMG, Universal Music, Warner Bros, Popular Science, Disney, and so on), non-profit organizations (amongst which are Amnesty International, the United Nations and Oxfam), schools (Harvard, MIT and many more), even governments (including the Belgian, French, U.S. and New Zealand) and important organisations such as NASA and NATO. The list is seemingly endless [7].

Drupal is also strongly focused on innovation, and always closely follows (or leads!) the cutting edge of the world wide web. The Drupal community even has a saying for this:

the drop is always moving [8]

This means there will always be an upgrade from one major Drupal core version to the next, but it will only preserve your data, your code will stop working. This is what prevents Drupal from having an excessive amount of legacy code that many other projects suffer from. Each new major version contains many, often radical, changes in the APIs.

Maturity

Indicators of project maturity are also present: Drupal has a set of coding standards [9] that must be followed strictly. For even the slightest deviation (a single missing space), a patch can be marked as 'needs work'. It also has a large security team [10] which releases security advisories whenever a security flaw is found in either Drupal core or any of the contributed modules.

Community

That brings us to the final part of this brief general introduction to Drupal: the gold of Drupal is in its community. The community is in general very eager to help getting newcomers acquainted with the ins and outs of Drupal. Many people have learned their way through the Drupal APIs by helping others (including myself). The result of this vibrant community is that there is a very large collection of more than 4000 modules [11] and more than 500 themes [12] available for Drupal, albeit of varying quality. This is what enables even the less technically adept to build a web site with complex interactions, without writing a single line of code.

4 Why it matters

Page loading performance matters for a single reason:

Users care about performance!

Your web site's visitors will not be timing the page loads themselves, but they will browse elsewhere when you are 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 will 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 [13]
- 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) [13]
- Google: 500 ms of extra load time caused 20% fewer searches [13]
- Google: trimming page size by 30% resulted in 30% more map requests [14]

It is 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 being 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 include web sites with the Update Status module enabled, there are over 140,000 web sites as of February 22, 2009, see [15]) and would therefore affect ever more people. Near the end of my bachelor thesis, on June 14, 2009, this had already grown to more than 175,000 web sites.
- Drupal is international, thanks to its internationalization and localization support, and thanks to that it is used for sites with very geographically dispersed 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.

5 Key Properties of a CDN

I will 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 is extremely hard to decide which CDN to use. In fact, by just looking at a CDN's performance, it is close to impossible [17, 18]!

That is why CDNs achieve differentiation through their feature sets, not through performance. 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 — this means that the CDN guarantees that it will be online 100% of the time.

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: replace your own domain name with the CDN's domain name. The CDN will then apply the *Origin Pull* technique and will periodically pull the files from the origin (that is your server). How often that is, depends on how you have configured headers (particularly the **Expires** header). 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 from your part 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 is 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 are 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 table.

My aim is to support the following CDNs in this thesis:

- any CDN that supports Origin Pull
- any CDN that supports FTP
- Amazon S3 [97] and Amazon CloudFront [98]. Amazon S3 (or Simple Storage Service in full) is a storage service that can be accessed via the web (via REST and SOAP interfaces). It is used by many other web sites and web services. It has a pay-per-use pricing model: per GB of file transfer and per GB of storage.

Amazon S3 is designed to be a storage service and only has servers in one location in the U.S. and one location in Europe. Recently, Amazon CloudFront has been added. This is a service on top of S3 (files must be on S3 before they can be served from CloudFront), which has edge servers everywhere in the world, thereby acting as a CDN.

6 Profiling tools

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

Lord Kelvin

The same applies to page loading performance: if you cannot measure it, you cannot know which parts have the biggest effect and thus deserve your focus. So before doing any real work, we will 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). [19]

So a list of tools will be evaluated: UA Profiler, Cuzillion, YSlow, Hammerhead, Apache JMeter, Gomez/Keynote/WebMetrics/Pingdom and Jiffy/Episodes. From this fairly long list, the tools that will be used while improving Drupal’s page loading performance will be picked, 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 to profile their site’s page loading performance.

6.1 UA Profiler

UA Profiler [20] is a crowd-sourced project for gathering browser performance characteristics (on the number of parallel connections, downloading scripts without blocking, caching, et cetera). The tests run automatically when you navigate to the test page from any browser – this is why it is powered by crowd sourcing.

It is a handy *reference* to find out which browser supports which features related to page loading performance.

6.2 Cuzillion

Cuzillion [21] was introduced [22] on April 25, 2008 so it is a relatively new tool. Its tag line, “*cuz there are zillion pages to check*” indicates what it is 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 is 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 is 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*.

Here is a simple example to achieve 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 3). This generates a unique URL. You can then copy this URL to all browsers you would like to test.

As you can see, Safari and Firefox behave very differently. In Safari (see figure 4), the loading of the first image seems to be deferred until the inline script has been executed (the images are displayed when the light purple bars become dark purple). In Firefox (see figure 5), 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 (the images are displayed when the gray bars stop). 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.

6.3 YSlow

YSlow [27] is a Firebug [25] extension (see figure 6) that can be used to analyze page loading performance through thirteen rules. These were part of the original fourteen rules [29] – of which there are now thirty-four – of “Exceptional Performance” [28], as developed by the Yahoo! performance team.

YSlow 1.0 can only evaluate these thirteen 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 does not test the latency or speed of a CDN, it just checks if you are using one. As an example, because

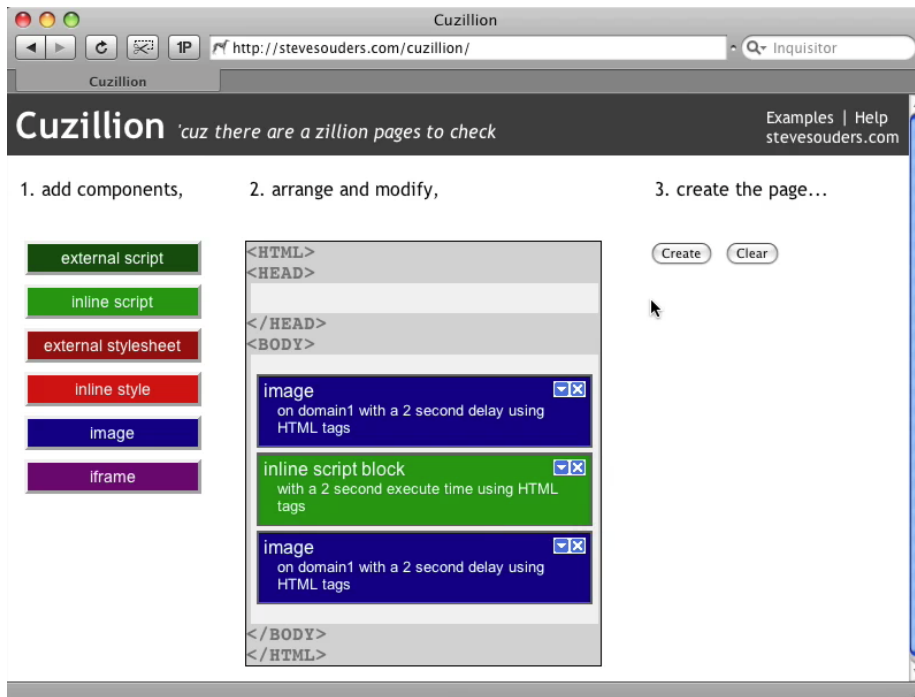


Figure 3: The example situation created in Cuzillion.

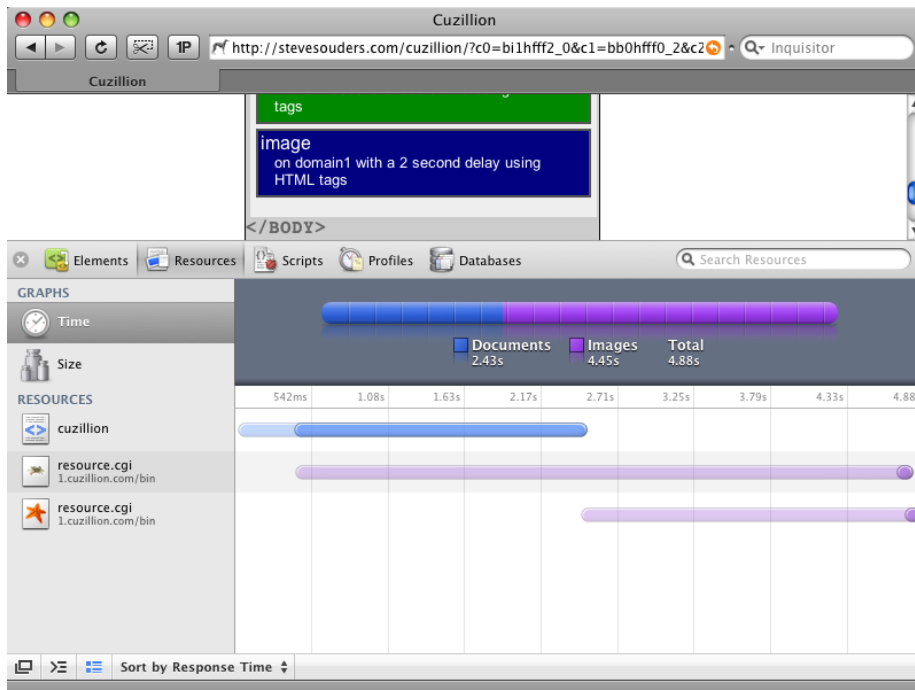


Figure 4: The example situation in Safari 3.

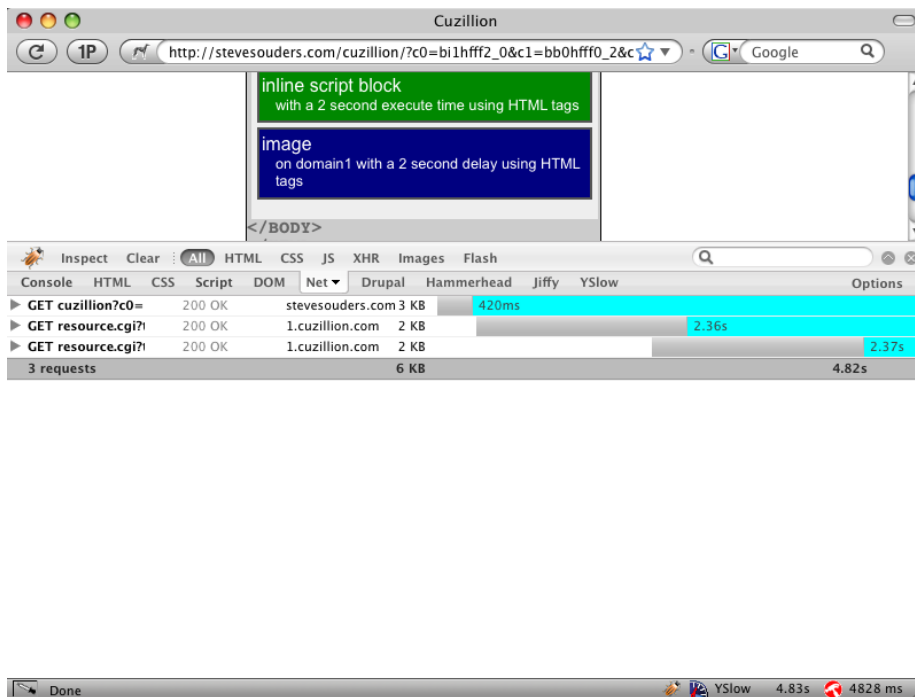


Figure 5: The example situation in Firefox 3.

you have 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 7.

That, and the fact that some of the rules it analyzes are only relevant to *very* big web sites. For example, one of the rules (#13, "Configure ETags") is only relevant if you are 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 [30]. YSlow 2.0 [31] 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 will stick with that. It is a very powerful and helpful tool as it stands, it will just get better. But remember the two caveats: it only verifies rules (it does not measure real-world performance) and some of the rules may not be relevant for your web site.

6.4 Hammerhead

Hammerhead [23, 24] is a Firebug [25] 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

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

Figure 6: YSlow applied to drupal.org.

Performance Grade: F (59)		Performance Grade: C (70)	
F	1. Make fewer HTTP requests ▾	F	1. Make fewer HTTP requests ▾
F	2. Use a CDN ▾	A	2. Use a CDN ▾
F	3. Add an Expires header ▾	F	3. Add an Expires header ▾
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 ▾	n/a	8. Make JS and CSS external ▾
A	9. Reduce DNS lookups	A	9. Reduce DNS lookups
A	10. Minify JS ▾	A	10. Minify JS ▾
A	11. Avoid redirects	A	11. Avoid redirects
A	12. Remove duplicate scripts	A	12. Remove duplicate scripts
F	13. Configure ETags ▾	F	13. Configure ETags ▾

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

Figure 7: Tricking YSlow into thinking drupal.org is using a CDN.

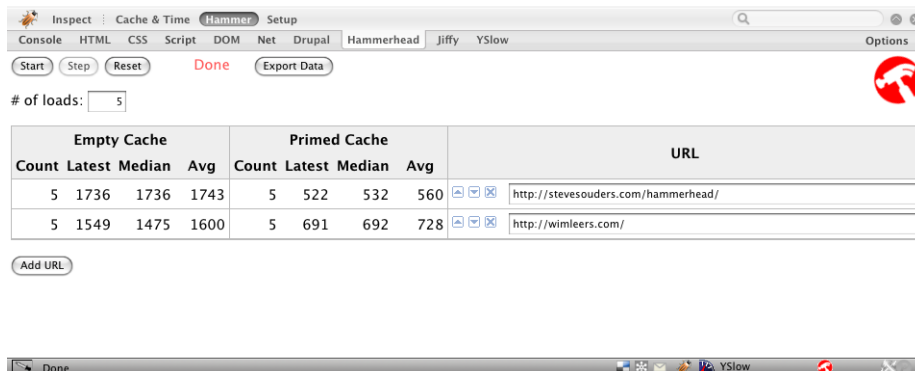


Figure 8: Hammerhead.

while you are working. It is far more effective to prevent page loading performance problems due to changes in code, because you have the test results within seconds or minutes after you have made these changes!

Of course, you could also use YSlow (see section 6.3) or FasterFox [26], but then you have to load the page multiple times (i.e. *hammer* the server, this is where the name comes from). And you would still have to set up the separate testing conditions for each page load that Hammerhead already sets up for you: 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 is 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. A screenshot of Hammerhead is provided in figure 8.

6.5 Apache JMeter

Apache JMeter [33] 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 is run, it does not give a good big picture.
- It is not an actual browser, so it does not download components referenced from CSS or JS files.
- Also because it is not an actual browser, it does not behave the same as browsers when it comes to parallel downloads.

- It requires more setup than Hammerhead (see section 6.4), so it is less likely that a developer will make JMeter part of his workflow.

It can be very useful in case you are 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 [34, 35]

6.6 Gomez/Keynote/WebMetrics/Pingdom

Gomez [36], KeyNote [37], WebMetrics [38] and Pingdom [39] are examples of third-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

6.6.1 Limited number of measurement points

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

6.6.2 No real-world browsers

Most, if not all of these services use their own custom clients [46]. That implies their results are not a representation of the real-world situation, which means you cannot 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 each site has different visitors and therefor also a different mix of browsers.

6.6.3 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 [40] 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 [49].

There are two ways to measure Web 2.0 web sites [50]:

1. *manual scripting*: identify timing points using scripting tools (Selenium [41], Keynote’s KITE [42], et cetera). This approach has a long list of disadvantages: low accuracy, high switching costs, high maintenance costs, synthetic (no real-world measurements).
2. *programmatic scripting*: timing points are marked by JavaScript (Jiffy [47], Gomez Script Recorder [43], et cetera). 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 would not have to reinvent the wheel every time and switching costs would be much lower. See the Jiffy/Episodes later on.

6.6.4 Paid & closed source

The end user is dependent upon the third party service to implement new instrumentations and analyses. It is typical for closed source applications to only implement the most commonly asked feature and because of that, the end user may be left out in the cold. There is a high cost for the implementation and a also a very high cost when switching to a different third party service.

6.7 Jiffy/Episodes

6.7.1 Jiffy

Jiffy [45, 46, 47] is designed to give you real-world information on what is 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 do not control all the components of your web site (e.g. widgets of photo and music web sites, contextual ads or web analytics services), it is important that you can monitor their performance. It overcomes four major disadvantages that were listed previously:

1. it can measure *every* page load if desired
2. real-world browsers are used, because it is 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 measuring your pages and reporting measurements
- Apache configuration: to receive and log measurements via a specific query string syntax
- Ingestor: parse logs and store in a database (currently only supports Oracle XE)
- Reporting toolset
- Firebug extension [48], see figure 9

Jiffy was built to be used by the WhitePages web site [44] 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 has been released as an open source project, but at the time of writing, the last commit was on July 25, 2008. So it is a dead project.

6.7.2 Episodes

Episodes [52, 53] is very much like Jiffy. There are two differences:

1. Episodes' goal is to become an industry standard. This would imply that the aforementioned third 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.

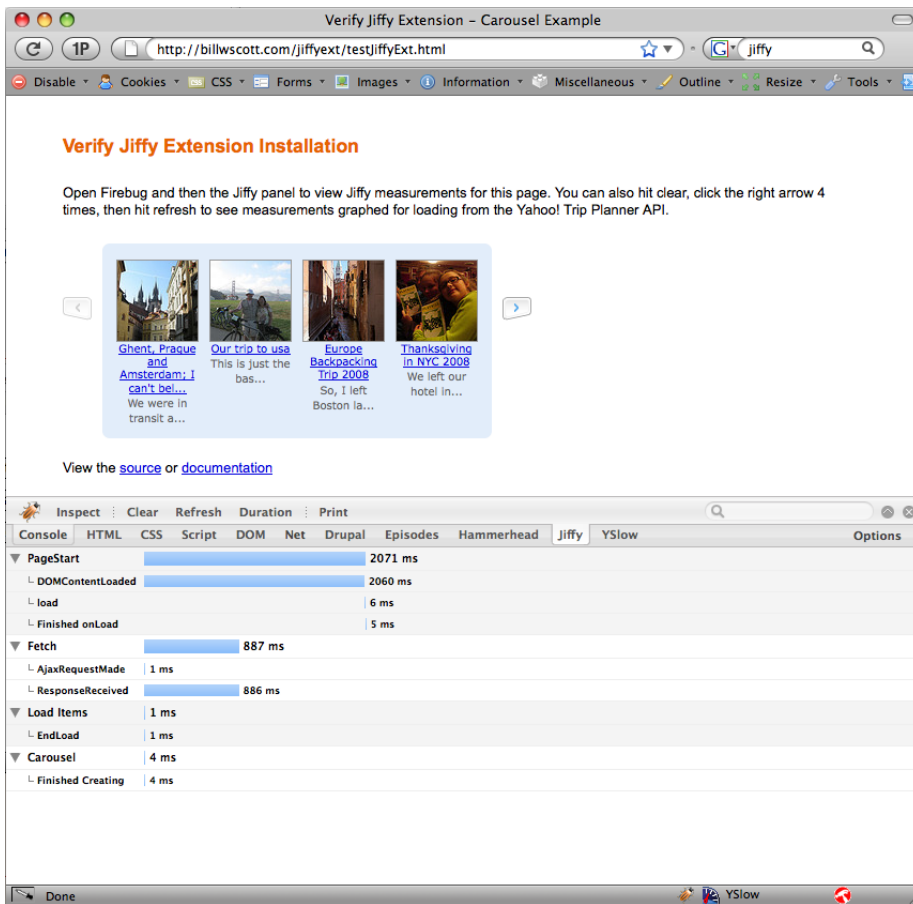


Figure 9: Jiffy.

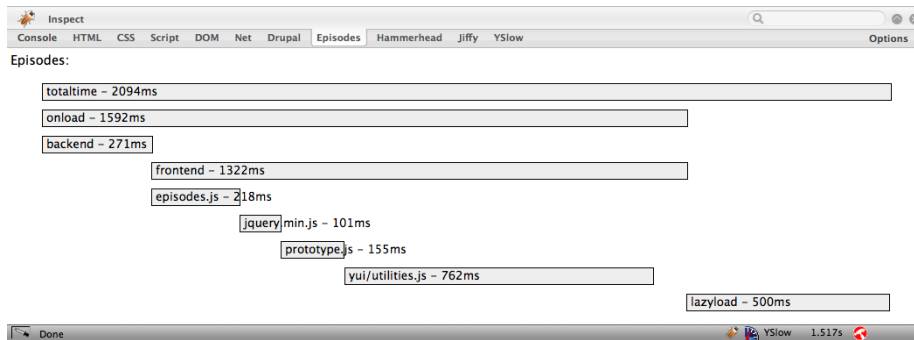


Figure 10: Episodes.

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 is 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 does not support `window.postMessage()`. These files are operational but have not had wide testing yet.
- It uses the same query string syntax as Jiffy uses to perform logging, which means Jiffy's Apache configuration, ingestor and reporting toolset can be reused, at least partially.
- It has its own Firebug extension, see figure 10.

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 full-time on all things related to page loading performance (which suggests we might see integration with Google's Analytics [51] service in the future). Add in the fact that Jiffy has not been updated since its initial release, and it becomes clear that Episodes is the better long-term choice.

6.8 Conclusion

There is not 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 is not necessarily meaningful in real-world situations, because it e.g. only checks if you are using a CDN, not how fast that CDN is. But 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 will perform in the next section.

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 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 have created this integration as part of my bachelor thesis, see section 8.

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 is 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 does not make much sense to just rephrase the article here in my thesis text, so instead I would like to forward you to that article [16] for the details. The article analyzes Drupal based on the 14 rules defined in Steve Souder's High Performance Web Sites book.

The essence of the article is that Drupal does some things right already, but many more not yet. The things Drupal did wrong then — and still does wrong today because nothing has changed in this area — yet:

- 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 dynamically alter file URLs is needed, but this is not 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 Improving Drupal: Episodes integration

The work I am doing as part of bachelor thesis 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 [54]. This module integrates the Episodes framework for timing web pages (see section 6.7.2) 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`) 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. That is the only modification you have to make by hand. It is acceptable because a theme always must be tweaked for a given web site.
- Provides basic reports with charts to make sense of the collected data.

I actually wrote two Drupal 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 or on the same. It provides basic reports. It is recommended to install this on a separate Drupal web site, and preferably even a separate web server, because it has to process a lot of data and is not optimized. That would have led me too far outside of the scope of this bachelor thesis.

You could also choose to not enable the Episodes Server module and use an external web service to generate reports, but for now, no such services yet exist. This void will probably be filled in the next few years by the business world. It might become the subject of my master thesis.

8.1 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.

The main measurement points are:

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>

```

- starttime: time of requesting the web page (when the `onbeforeunload` event fires, the time is stored in a cookie); not in the HTML file
- 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, including lazily-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 is possible to also measure the time it took to load the CSS (lines 8-9, this would be the `css` episode) and JavaScript files in the header (line 10, this would be the `headerjs` episode) and in the footer (line 27, this would be the `footerjs` episode), for example. It is possible to measure just about anything you want.

For a visual example of all the above, see figure 13.

8.2 Making episodes.js reusable

The `episodes.js` file provided at the Episodes example [55] is in fact just a rough sample implementation, an implementation that indicates what it should look like. It contained several hardcoded URLs, does not measure the sensible default episodes, contains a few bugs. In short, it is an excellent and solid start, but it needs some work to be truly reusable.

There also seems to be a bug in Episodes when used in Internet Explorer 8. It is actually a bug in Internet Explorer 8: near the end of the page loading sequence, Internet Explorer 8 seems to be randomly disabling the `window.postMessage()` JavaScript function, thereby causing JavaScript errors. After a while of searching cluelessly for the cause, I gave up and made Internet Explorer 8 also use the backwards-compatibility script (`episodes-compat.js`), which overrides the `window.postMessage()` method. The problem had vanished. This is not ideal, but at least it works reliably now.

Finally, there also was a bug in the referrer matching logic, or more specifically, it only worked reliably in Internet Explorer and intermittently worked in Firefox, due to the differences between browsers in cookie handling. Because of this bug, many `backend` episodes were not being measured, and now they are.

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:

```
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>
```

This way, you can initialize the variables to the desired values without customizing `episodes.js`. Line 6 should be as early in the page as possible, because it is the most important reference time stamp.

8.3 Episodes module: integration with Drupal

8.3.1 Implementation

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

- Implemented `hook_install()`, through which I set a module weight of -1000. This extremely low module weight ensures the hook implementations of this module are always executed before all others.
- 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 is necessary to make Episodes work (see section 8.2). 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 have implemented, all JavaScript code can be measured through Episodes without hacking Drupal core. A simplified version of what it does can be seen here:

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);  
  }  
};
```

- Some of the Drupal behaviors are too meaningless to measure, so it would be nice to be able to mark some of the behaviors as ignored. That is 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 [56]. 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 [58].

- For measuring the `css` and `headerjs` episodes, it is necessary to make a couple of simple (copy-and-paste) changes to the `page.tpl.php` of the Drupal theme(s) you are 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 is recommended, but not required.
- And of course a configuration UI (see figure 11 and figure 12) using the Forms API [57]. It ensures the logging URL (this is the URL through which the collected data is logged to Apache’s log files) exists and is properly configured (i.e. returns a zero-byte file).

8.3.2 Screenshots

The screenshot shows the 'Episodes' settings page in a Drupal administration interface. The breadcrumb trail is 'Home > Administer > Site configuration'. The page has two tabs: 'Settings' (active) and 'Behaviors'. Under the 'Settings' tab, there are three sections: 'Status', 'Logging', and 'Advanced settings'. The 'Status' section has three radio buttons: 'Disabled', 'Debug mode', and 'Enabled' (selected). Below it is a paragraph explaining the options. The 'Logging' section has a paragraph explaining logging, a 'Logging status' section with 'Disabled' and 'Enabled' (selected) radio buttons, and a 'Logging URL' section with a text input field containing 'http://example.com/episodes/beacon' and a paragraph explaining the URL. The 'Advanced settings' section is expanded, showing an 'Excluded paths' section with a text area containing 'admin/*' and a paragraph explaining the format. At the bottom 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.

Logging

Without logging, the collected episodes will only be visible in the Episodes Firebug plug-in in Firefox. By logging the collected statistics, the Episodes Server module can be used to visualize the collected statistics.

Logging status:

Disabled

Enabled

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

Logging URL:

The logging URL (also called *beacon URL*) you would like to use. Can also be the URL of a different server, or potentially even a web service that analyzes the logs for you.

▼ **Advanced settings**

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.

Figure 11: 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.**

[Scan](#)

Ignored behaviors:

viewsDependent (views) [Add](#)

All ignored behaviors

collapse (core)	Remove
multiselectSelector (core)	Remove
viewsDependent (views)	Remove

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

[Save](#)

Figure 12: Episodes module behaviors settings form.

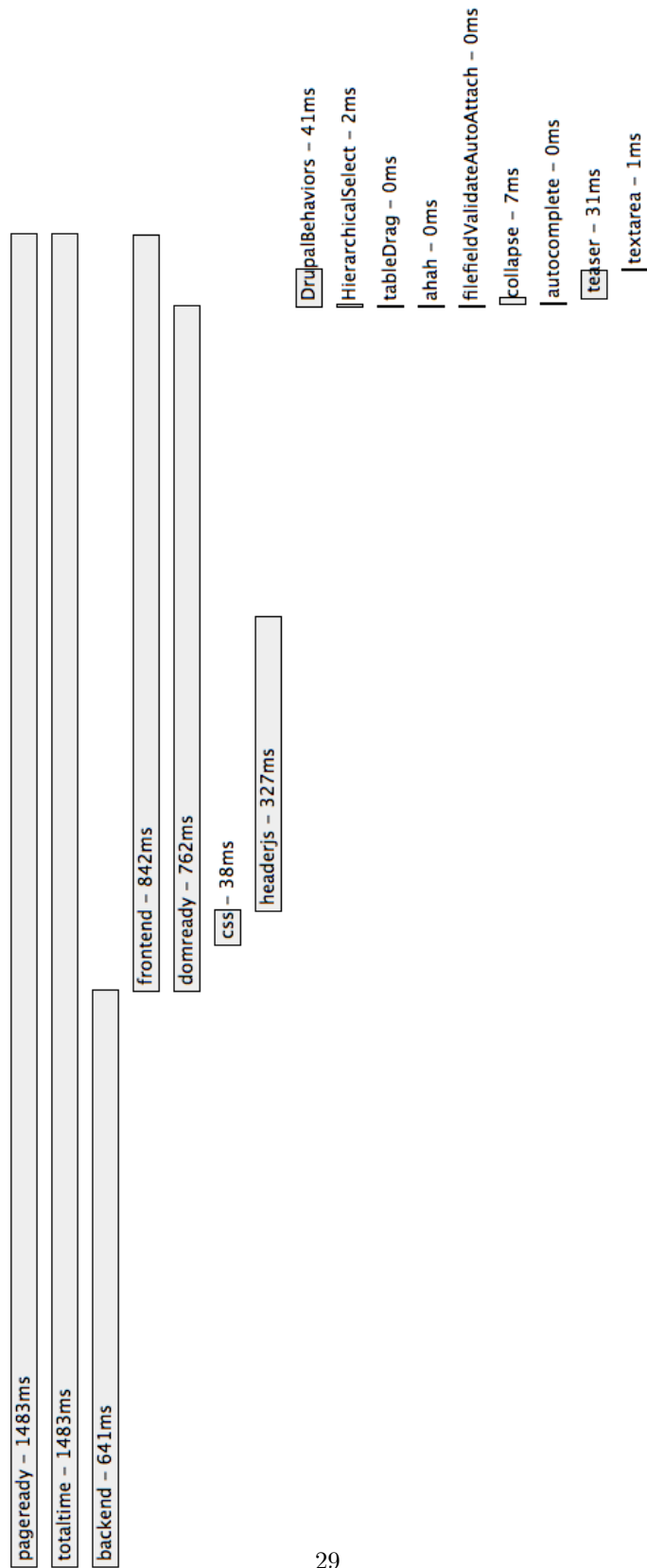


Figure 13: Results of Episodes module in the Episodes Firebug add-on.

8.4 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.

8.4.1 Implementation

- First of all, the Apache HTTP server is a requirement as this application's logging component is used for generating the log files. Its logging component has been proven to be scalable, so there is no need to roll our own. The source of this idea lies with Jiffy (see section [6.7.1 on page 16](#)).
- 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 completely rewrote it to obtain 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 [\[60\]](#). Also, IPs are converted to country codes using the `ip2country` Drupal module [\[61\]](#). This is guaranteed to work thanks to the included meticulous unit tests.
- For the reports, I used the Google Chart API [\[59\]](#). You can see an example result in figures [15](#), [16](#) and [17](#). It is possible to compare the page loading performance of multiple countries by simply selecting as many countries as you would like in the "Filters" fieldset.
- And of course again a configuration UI (see figure [14](#)) using the Forms API [\[57\]](#). It ensures the log file exists and is accessible for reading.

8.4.2 Screenshots

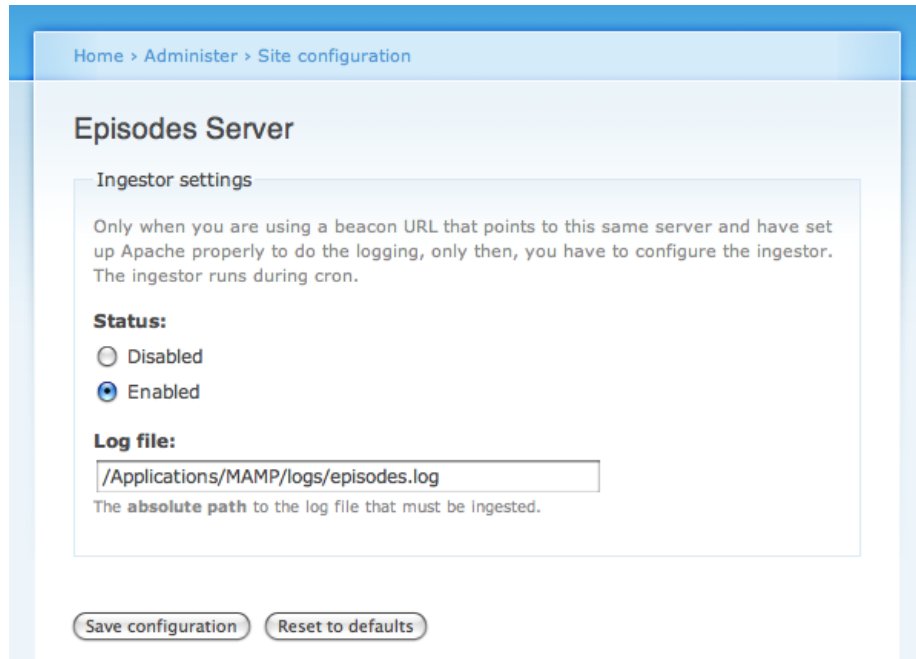


Figure 14: Episodes Server module settings form.

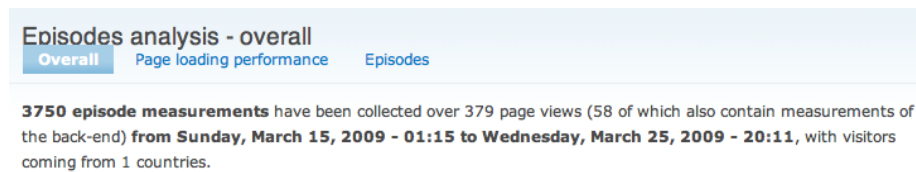


Figure 15: Episodes Server module: overall analysis.

8.4.3 Desired future features

Due to lack of time, the basic reports are ... well ... very basic. It would be nice to have more charts and to be able to filter the data of the charts. In particular, these three 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

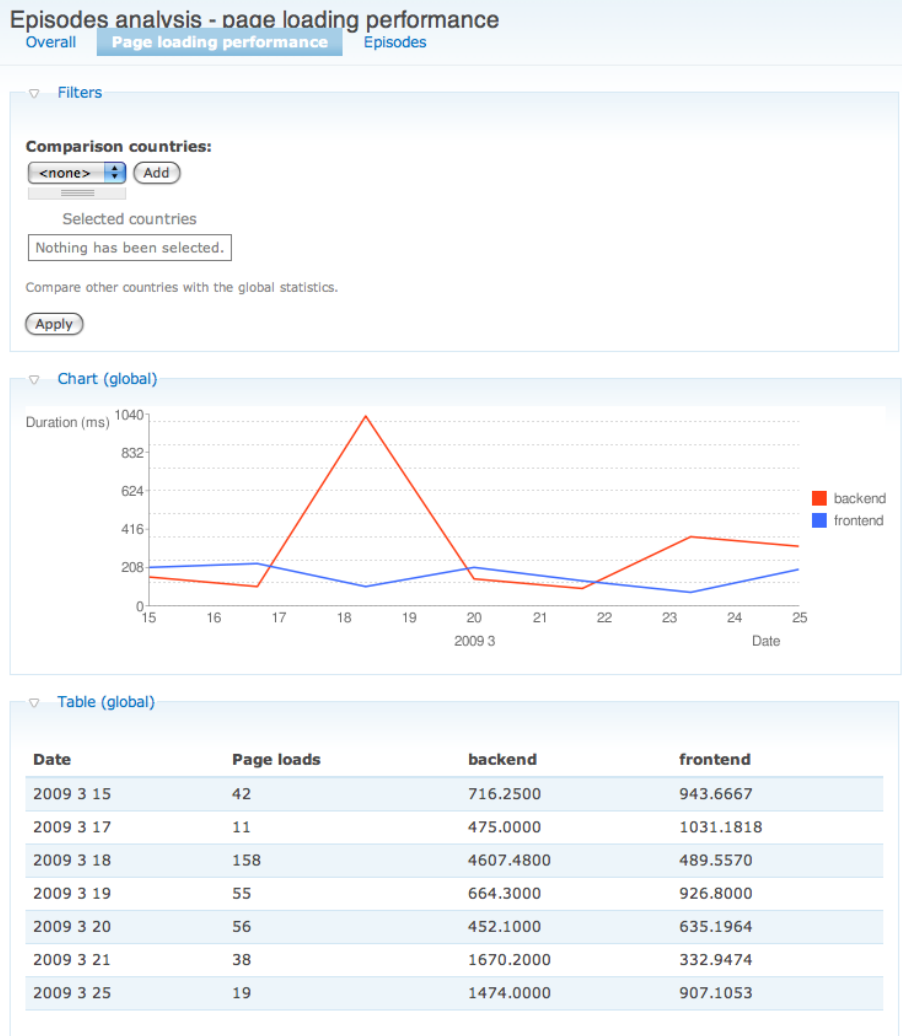


Figure 16: Episodes Server module: page loading performance analysis.

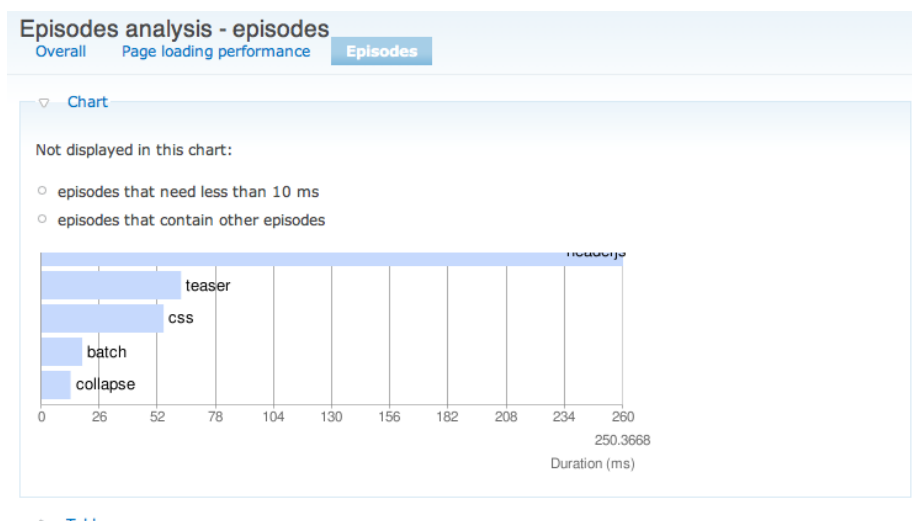


Figure 17: Episodes Server module: episodes analysis.

8.5 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. Thus, I simply kept Steve Souder's `onbeforeunload` method to log the time when a next page was requested. The major disadvantage of this method is that it is impossible to measure the `backend` episode for each page load: it is only possible to measure the `backend` episode when the user navigates through our site (more specifically, when the referrer is the same as the current domain).
 - 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 cannot *relate* it to the measurements in the browser. I implemented this using 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 therefore 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 would not be better. The real solution is to use a different server to run the Episodes Server module on or even an external web service. Better even is to log to your own server and then send the logs to an external web service. This way you stay in control of all your data! Because you still have your log data, you can switch to another external web service, thereby 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. As I already mentioned before, writing an advanced profiling tool could be a master thesis on its own.

8.6 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, which suggests that it will probably be worthwhile/helpful for the Drupal community as well.

Unfortunately for me, Steve Souders is a very busy man, speaking at many web-related conferences, teaching at Stanford, writing books and working at Google. He did not manage to get back to the questions I asked him.

9 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. Next, we will examine how a web site can take advantage of a CDN.

As explained in section 5, 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 is necessary to be able to transparently switch between pull and any of the transfer protocols for push. Suppose that you are using CDN A, which only supports FTP. when you want to switch to a cheaper, yet better CDN B, that would be a costly operation, because CDN B only supports a custom protocol.

To further reduce costs, it is necessary that we can do the preprocessing ourselves (be that video transcoding, image optimization or anything else). Also note that many CDNs do not support processing of files — but it can reduce the amount of bandwidth consumed significantly, and thereby the bill received every month.

That is 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. install 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 (so you can alter the existing URLs to files to the ones for the CDN)

9.1 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 is more to it than just that, so here is 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 (and sync) new files instantly: through inotify on Linux, FSEvents on Mac OS X and the FindFirstChangeNotification API or ReadDirectoryChanges API on Windows (there is 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 that it was still syncing and that have been added, modified and deleted while it was not running
- 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)
- All the logic of the daemon should be contained in a single module, to allow for quick refactoring.

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 may be downloading files directly from the web server instead of from the CDN. This increases the load on the web server unnecessarily and also increases the page load time for the visitors.

For example, one visitor has uploaded images as part of the content he created. All visitors will be downloading the image from the web server, which is suboptimal, considering that they could have 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 customer base, you may either want to pay for a global CDN or a local one. If you are a global company, a global CDN makes sense. But if you are present only in a couple of countries, say the U.S.A., Japan and France, it does not make sense to pay for a global CDN. It is 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 is 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 have entered 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

The daemon, or at least one or more of the modules that were written for it, can be reused in other applications. For example:

- Back-up tool
- Video transcoding server (e.g. to transcode videos uploaded by visitors to H.264 or Flash video)
- Key component in creating your own CDN
- Key component in a file synchronization tool for consumers

9.2 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 is too hard to configure, nobody will use it. If it is 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. Beneath the config root node, there are 3 child nodes, one for each 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. **sources** itself also has an optional **ignoredDirs** attribute, which will subsequently be applied to all **filter** nodes. While unnecessary, this prevents needless duplication of **ignoredDirs** nodes inside **filter** nodes.
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 does not 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 is 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 9.3.1) 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 would like to use. They will 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 (on the server) inside which the files will be transported.

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

9.3 Python modules

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

9.3.1 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

Listing 3: Sample configuration file.

```

<?xml version="1.0" encoding="UTF-8"?>
<config>
  <!-- Sources -->
  <sources ignoredDirs="CVS:.svn">
    <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>
      </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>

```

a given file path will result in a positive match.

The five conditions that can be set (as soon as one or more conditions are set, **Filter** will work) 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) **threshold**: the threshold in bytes

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

9.3.2 pathscanner.py

As is to be expected, this module provides the `PathScanner` class, which scans paths and stores them in a SQLite [68] 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. This mean `PathScanner` is used during the initialization of the daemon, regardless of the available file system monitors.

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 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. Changes are committed to the database on a per-directory level. However, if many changes occurred in a single directory and if every change would be committed separately, the concurrency level would rise unnecessarily. By default, every batch of 50 changes inside a directory is committed.

This class provides you with 8 methods:

- `initial_scan()` to build the initial database – works recursively
- `scan()` to get the changes – does not 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)

Special care had to be taken to not scan directory trees below directories that are in fact symbolic links. This design decision was made to mimic the behavior of file system monitors, which are incapable of following symbolic links.

This module does not have any tests yet, because it requires *a lot* of mock functions to simulate system calls. It has been tested manually thoroughly though.

9.3.3 fsmonitor.py

This time around, there is 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, easy to use abstraction for each major operating system’s file system monitor:

- Uses `inotify` [62] on Linux (kernel 2.6.13 and higher)
- Uses `FSEvents` [64, 65] on Mac OS X (10.5 and higher)
- Falls back to polling when neither one is present

Windows support is possible, but has not been implemented yet due to time constraints. There are two APIs to choose between: `FindFirstChangeNotification` and `ReadDirectoryChanges`. There is a third, the `FileSystemWatcher` class, but this is only usable from within `.NET` and `Visual C++`, so it is an unlikely option because it is not directly accessible from within Python. This was already mentioned in 9.1.

The `ReadDirectoryChanges` API is more similar to `inotify` in that it triggers events on the file level. The disadvantage is that this is a blocking API. `FindFirstChangeNotification` is a non-blocking API, but is more similar to `FSEvents`, in that it triggers events on the directory level. A comprehensive, yet concise comparison is available at [66].

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 was not 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 does not maintain a complete history of events. And that is understandable: it is impossible to maintain a history of *every* file system event. Over time, there would not 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 would be too resource-intensive to detect, or at the very least it would not scale. On the plus side, `FSEvents` maintains a complete history of events. `PathScanner`'s `scan()` method is used to detect the changes to the files in each directory that was changed.

Implementations that do not 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 9.3.2) 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 is just slower. But it is sufficient for now. This was not implemented due to time constraints.

To implement this, one would simply need to get the highest `mtime` (modification time of a file) stored in the database, and then ask `FSEvents` to send the events from that starting time to our callback function, instead of starting from the current time.

9.3.4 `persistent_queue.py` and `persistent_list.py`

In order to provide data persistency, I wrote a `PersistentQueue` class and a `PersistentList` class. 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 data structures in a logical manner (i.e. related persistent data structures 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 threshold can be configured), but for `PersistentList` there is no such restriction: it is cached in-memory in its entirety. It is not designed for large datasets, but `PersistentQueue` is.

`PersistentQueue` is used to store the events that have been triggered by changes in the file system via `fsmonitory.py`: (`input_file`, `event`) pairs are stored in a persistent queue. Because the backlog (especially the one after the initial scan) can become very large (imagine 1 million files needing to be synced), this was a necessity.

`PersistentList` is used to store the files that are currently being processed and the list of files that have failed to sync. These must also be stored persistently, because if the daemon is interrupted while still syncing files, these lists will not be empty, and data could be lost. Because they are persistent, the daemon can add the files in these lists to the queue of files to be synced again, and the files will be synced, as if the daemon was never interrupted.

The first question to arise is: “Why use SQLite in favor of Python’s built-in `shelve` module [72]?” Well, the answer is simple: aside from 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` is not 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 will not get corrupt?” The answer is: we can not. But the same applies to Python’s `shelve` module. However, the aforementioned advantages of SQLite give plenty of reasons to choose SQLite over `shelve`. Plus, SQLite is thoroughly tested, even against corruption [69]. It is also used for very large datasets (it works well for multi-gigabyte databases but is not designed for terabyte-scale databases [70]) and by countless companies, amongst which Adobe, Apple, Google, Microsoft and Sun [71]. So it is the best bet you can make.

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

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

9.3.5 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 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, checking if the processor (and therefor the entire chain of processors) should be run per server or not 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 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 and are executed before a file is synced, most often to reduce the size of the file and thereby to decrease the page loading time.

They are also allowed to change the base name of the input file, but they are not allowed to change its path. This measure was taken to reduce the amount of data that needs to be stored to know which file is stored where exactly in the database of synced files (see later: section 9.4). This is enforced *by convention*, because in Python it is 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 one is the input file of processor two, 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 (described later on in this section), in which 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 do not 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
```

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 already, that is 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 is 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 time stamp 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).

image_optimizer.py

This processor module is inspired by [74]. It optimizes images losslessly, i.e. it reduces the filesize without touching the quality. The research necessary was not 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 [75, 76, 77, 78] at the Yahoo! user interface blog.

For GIF files, a conversion to PNG8 is performed using ImageMagick's [79] `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. The alpha channels of true color PNG (PNG24 & PNG32) 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` [80] 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 [78] have pointed out that there is a 75% chance that the JPEG file is best saved as baseline when it is smaller than 10 KB. For JPEG files larger than 10 KB, it is 94% likely that progressive JPEG will result in a better compression ratio. That is 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` [81].

Finally, animated GIF files can be optimized by stripping the pixels from each frame that do not change from the previous to the next frame. I use `gifsicle` [82] to achieve that.

There is one important nuance though: stripping metadata may also remove the copyright information, which may have legal consequences. So it is not recommended to strip metadata when some of the photos being hosted have been bought, which may be the situation for a newspaper web site, for example.

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 the CSS file uses relative URLs that are relative to the document root to reference images (or other media).
For example:
– `http://example.com/static/css/style.css`
becomes
– `http://cdn.com/example.com/static/css/style.css`
and its referenced file
– `http://example.com/static/images/background.png`
becomes
– `http://cdn.com/example.com/static/images/background.png`
after syncing. If the `style.css` file on the original server references `background.png` through the relative URL `/static/images/background.png`, then the `CSSURLUpdater` processor must be used to update the URL. Otherwise this relative URL would become invalid, since the correct relative URL for the CSS file on the CDN to reference has changed (because the base path has changed).
- The base names of the referenced files changes.
For example:
– `http://example.com/static/css/style.css`
becomes
– `http://cdn.com/example.com/static/css/style_1242440815.css`
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` [83] 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.

The `CSSURLUpdater` processor requires to be run per-server (and therefore the entire processor chain which it is part of), because it wants the referenced files (typically images, but possibly also fonts) to be on the same server.

All it 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 9.3.5) 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. It accepts CSS and JS files and runs the YUI Compressor [84] on them, which are then compressed by stripping out all whitespace and comments. For JavaScript, it relies on Rhino [85] to tokenize the JavaScript source, so it is very safe: it will not 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 [86]. For CSS (which is supported since version 2.0) it uses a regular-expression based CSS minifier.

9.3.6 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 [88]. 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.

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)
```

`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.

Django is a high-level Python Web framework that encourages rapid development and clean, pragmatic design. It is growing very strong in popularity. It is very different from Drupal, which aims to be both a CMS (install it and you have a working web site) and a framework. Django just aims to be a framework. It has APIs for many things, ranging from caching, database abstraction, forms to sessions, syndication, authentication and of course storage systems. I have extract that single API (and its dependencies) from Django and am reusing it.

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

1. Since Django is a well-known, widely used open source project with many developers and is powering many web sites, it is 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` [89] 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 [100] is an open source distributed file system
- (c) `CouchDBStorage`; Apache CouchDB [101] is a distributed, fault-tolerant and schema-free document-oriented database accessible via a RESTful HTTP/JSON API
- (d) `CloudFilesStorage`; Mosso/Rackspace Cloud Files [90], an alternative to Amazon S3
- (e) `S3Storage`; uses the official Amazon S3 Python module
- (f) `S3BotoStorage`; uses the boto [99] module to access Amazon S3 [97] and Amazon CloudFront [98]
- (g) `FTPStorage`; uses the `ftplib` Python module [96]

For the last two, transporters are available. The first three are not so widely used and thus not yet implemented, although it would be very easy to support them, exactly because all that is necessary, is to write thin wrappers. For the third, a transporter was planned, but scrapped due to time constraints. The fourth is not very meaningful to use, since the fifth is better (better maintained and higher performance).

The `CouchDBStorage` custom storage class was added about a month after I selected `django-storages` as the package I would use, indicating that the project is alive and well and thus a good choice.

The `CloudFilesStorage` custom storage class is part of `django-storages` thanks to my mediations. An early tester of the daemon requested [91] support for Cloud Files and interestingly enough, one day before, somebody had published a Python package (`django-cumulus` [92]) that did just that. Within a week, a dependency on `django-storages` was added from `django-cumulus` and `django-storages` had Cloud Files support. This is further confirmation that I had made a good choice: it suggests that my choice for `django-storages` is beneficial for both projects.

So I clearly managed to make a big shortcut (although it had to be made working outside of Django itself) to achieve my goal: supporting CDNs that rely on FTP or origin pulling (see section 5), as well as the Amazon S3 and Amazon 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 “do not 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: symbolic links. Although available only on UNIX, it is 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 have contributed three patches to `django-storages`:

1. `FTPStorage`: saving large files + more robust `exists()` [93]

- (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 has been tested with two different FTP servers and so far it works without problems with the following FTP servers, whereas it did not work with any of them before:
 - i. Xlight FTP Server 3.2 (used by SimpleCDN)
 - ii. Pure-FTPd (used by Rambla)

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

2. `S3BotoStorage`: set `Content-Type` header, fixed the setting of permissions, use HTTP and disable query authentication by default [94]

- (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 mime-type is set: `application/octet-stream`. This causes browsers to download files instead of displaying them.
- (b) The ACL (i.e. file permissions) 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 (which implies ridiculously long URLs will be generated) and HTTPS is used instead of HTTP, thereby preventing browsers from caching files. I have 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 [95]

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

davidbgk@wimleers Impressive patches, I will 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).

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 (see section 5).

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.

9.3.7 config.py

This module contains just one class: `Config`. `Config` can load a configuration file (parse the XML) and validate it. Validation does not happen through an XML schema, but through “manual” validation. The `filter` node is validated through the `Filter` class to ensure it is error free (a `Filter` object is created and the conditions from the `filter` node are set and when no exceptions are

raised, the conditions are valid). 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 is not – yet.

9.3.8 `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 is also how for example the Apache HTTP server does it [102]. 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 is running, the pid is written to the pid file every sixty seconds, in case the file is deleted accidentally.

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 is not much code). Having used it hundreds of times, it did not fail once, so it is reliable enough.

9.4 Putting it all together: `arbitrator.py`

9.4.1 The big picture

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

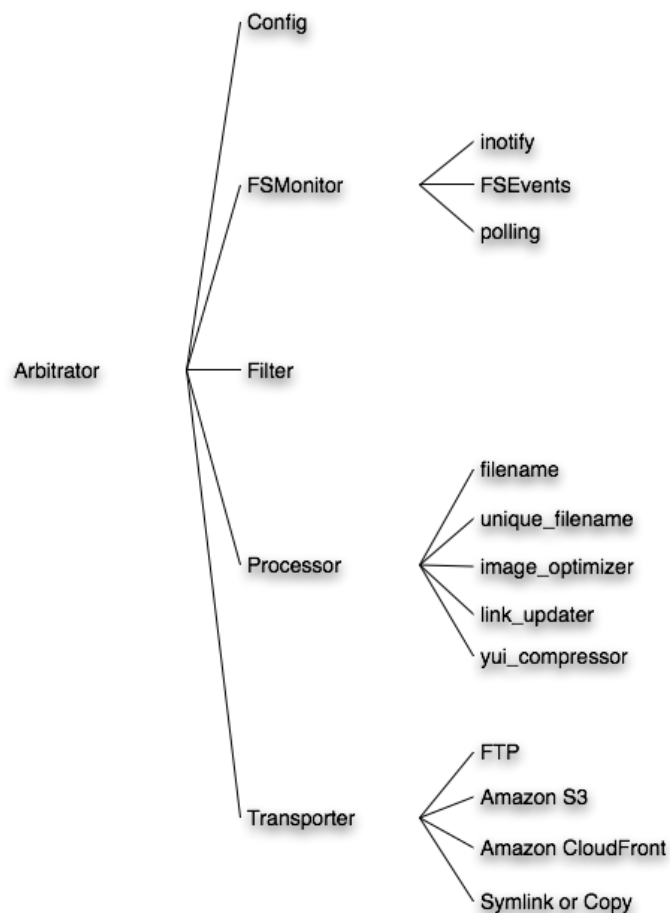


Figure 18: The big picture

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 us examine how exactly `Arbitrator` controls all components, and what happens before the `main` function.

9.4.2 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 is 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

- (a) create transporter pools (analogous to worker thread pools) for each server. These pools remain empty until transporters are necessary, because transporters are created whenever they are deemed necessary.
- (b) collect all metadata for each rule
- (c) initialize all data structures for the pipeline (queues, persistent queues and persistent lists)
- (d) move files from the 'files in pipeline' persistent list to the 'pipeline' persistent queue
- (e) move files from the 'failed files' persistent list to the 'pipeline' persistent queue
- (f) create a database connection to the 'synced files' database
- (g) initialize the file system monitor (`FSMonitor`)

2. run

- (a) start the file system monitor
- (b) start the processing loop and keep it running until the thread is being stopped
 - i. process the *discover* queue
 - ii. process the *pipeline* queue
 - iii. process the *filter* queue
 - iv. process the *process* queue
 - v. process the *transport* queues (1 per server)
 - vi. process the *db* queue
 - vii. process the *retry* queue
 - viii. allow retry (move files from the 'failed files' persistent list to the 'pipeline' persistent queue)
 - ix. sleep 0.2 seconds
- (c) stop the file system monitor

- (d) process the discover queue once more to sync the final batch of files to the persistent pipeline queue
- (e) stop all transporters
- (f) log some statistics

That is *roughly* the logic of the daemon. It should already make some sense, but it is likely that it is not yet clear what all the queues are for. And how they are being filled and emptied. So now it is time to learn about the daemon's *pipeline*.

9.4.3 Pipeline design pattern

This design pattern, which is also sometimes called “Filters and Pipes” [103, 104, 105, 106], is slightly under documented, but it is 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 19 while reading. Note that this figure does not 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 are 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 are removed from the 'files in pipeline' persistent list and we are 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 is a queue for each stage in the pipeline, plus some additional ones because the persistent data structures 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.

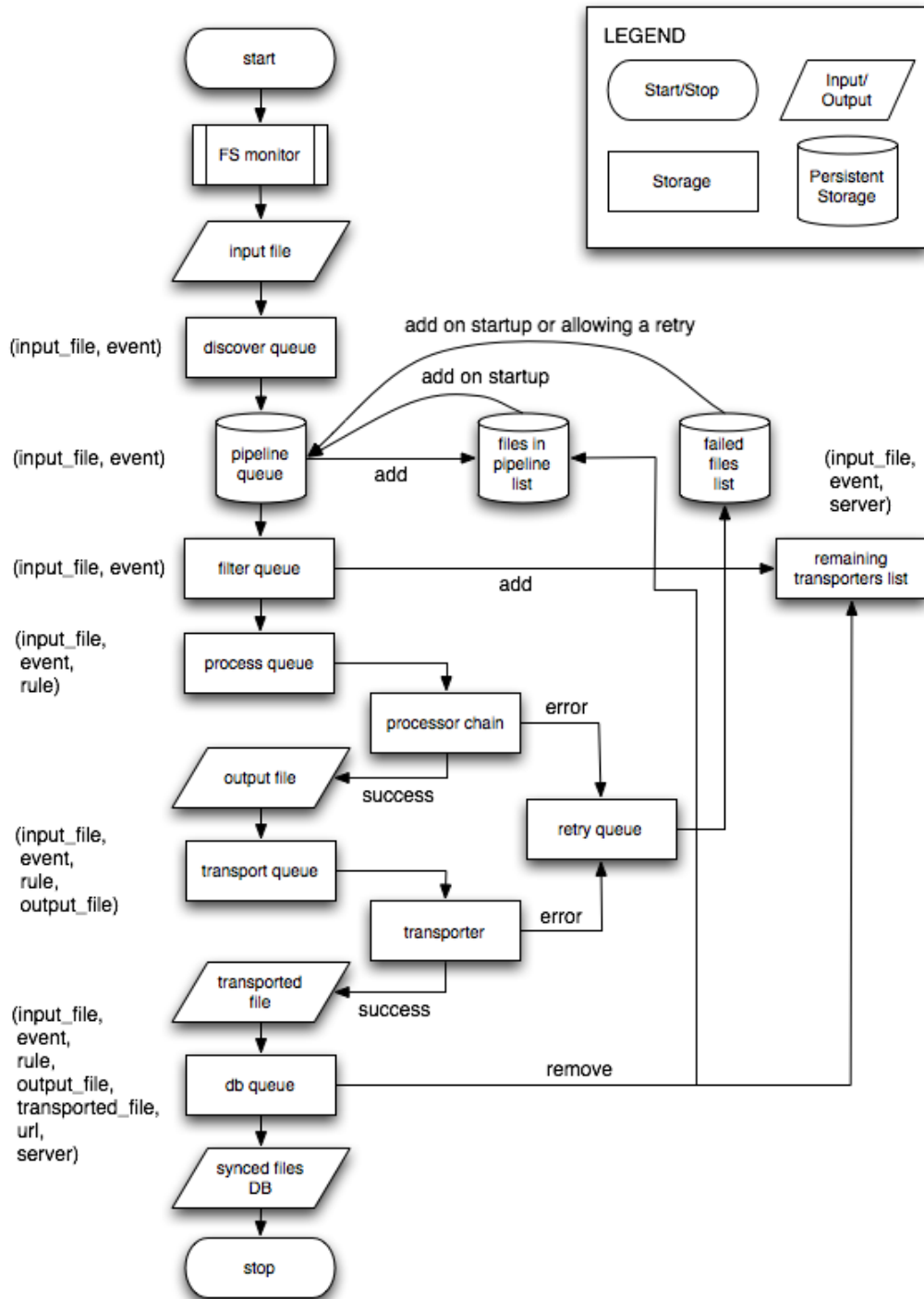


Figure 19: Flowchart of the daemon's pipeline.

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):

- 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 have 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 have found this in the form of currying [107]. 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 will 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`).

9.5 Performance tests

I have performed fairly extensive tests on 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 cross-platform functioning without problems 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 does not and should not 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 bug fixing 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.

9.6 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 third 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.

9.7 Desired future features

- Polling the daemon for its current status (number of files in the queue, files in the pipeline, processors running, transporters running, et cetera)
- 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.
- 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.

10 Improving Drupal: CDN integration

It should be obvious by now that we still need a module to integrate Drupal with a CDN, as Drupal does not 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.

10.1 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, it must also be easy to use the daemon for a Drupal web site. The former is called *basic mode* and the latter is called *advanced mode*, thereby indicating that the latter is more complex to set up (i.e. it requires you to set up the daemon). Here are the goals again, this time in more detail:

- 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 a 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 will 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 synced 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

10.2 Drupal core patch

I had the chance to speak to Andrew “drewish” Morton at DrupalCon DC about the Drupal core patch that is necessary for the CDN integration module for Drupal to become possible. 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.

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 are not 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 returns `FALSE` (meaning that the CDN cannot or should not 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.

This patch was also ported to Drupal 7 (which will be the next version of Drupal) and submitted for review. Unit tests were added (this is a requirement). The reviews are very positive so far (with Dries Buytaert, the Drupal founder, simply commenting “Awesome.” and adding it to his list of favorite patches) but it was submitted too late to ensure it got committed before this thesis text had to be finalized. However, the positivity of the reviews suggests that it is very likely that the patch will get committed.

10.3 Implementation

- A simple configuration UI was created using the Forms API [57]. Advanced mode cannot be started if the daemon is not 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 is 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 synced 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 is also PDO [108], which makes this sufficiently easy.

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

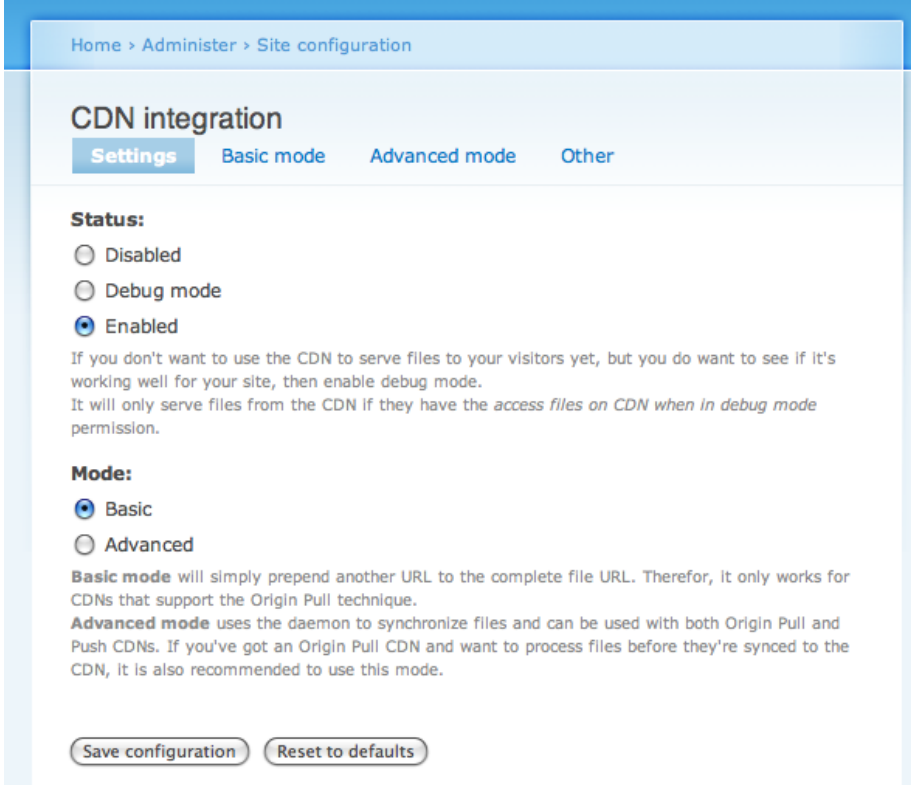
10.4 Comparison with the old CDN integration module

In January 2008, I wrote the initial version of the CDN integration module. It was written for Drupal 5 instead of Drupal 6 though and was pure PHP code, and thus limited by PHP's constraints. It did not 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 did not 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.

10.5 Screenshots

The configuration UI



The screenshot shows a web interface for configuring the CDN integration module. At the top, there is a breadcrumb trail: "Home > Administer > Site configuration". Below this, the main heading is "CDN integration". Underneath the heading are four tabs: "Settings" (which is active and highlighted in blue), "Basic mode", "Advanced mode", and "Other".

The "Settings" tab contains the following configuration options:

- Status:** Three radio button options: "Disabled", "Debug mode", and "Enabled". The "Enabled" option is selected.
- Mode:** Two radio button options: "Basic" and "Advanced". The "Basic" option is selected.

Below the radio buttons, there is explanatory text for the "Debug mode" and "Advanced mode" options. At the bottom of the form, there are two buttons: "Save configuration" and "Reset to defaults".

Figure 20: 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 21: 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 22: CDN integration module advanced mode settings form.

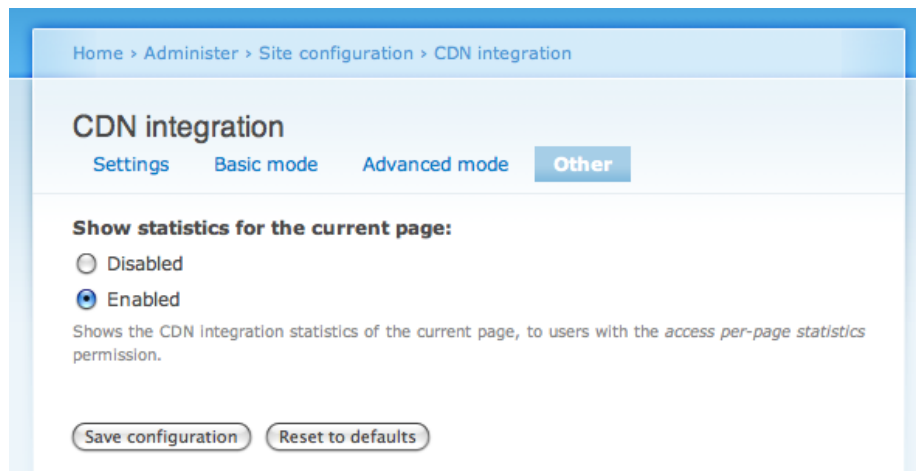


Figure 23: CDN integration module other settings form.

The status report

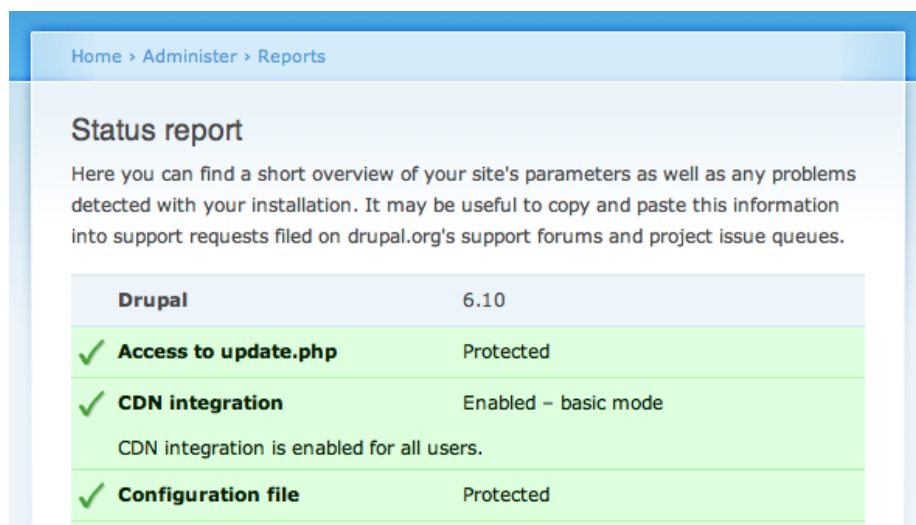


Figure 24: Status report (basic mode, enabled).

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.

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

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

Figure 25: 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.

Drupal	6.10
✓ Access to update.php	Protected
⚠ CDN integration	Disabled

CDN integration is disabled for all users.

Figure 26: 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.

Drupal	6.10
✓ Access to update.php	Protected
✗ CDN integration	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 27: 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.

Drupal	6.10
✓ Access to update.php	Protected
✓ CDN integration	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 28: 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 29: Per-page statistics.

11 Used technologies

- Languages
 - PHP
 - JavaScript
 - Python
 - SQL
- Frameworks
 - Drupal (Forms API, Batch API, menu system, Schema API, et cetera)
 - jQuery
 - Episodes [52]
 - Django's [87] `Storage` class [88] and its dependencies
- APIs/libraries
 - Browser.php [60]
 - Google Chart API [59]
 - FSEvents [64, 65] (through the Python-Objective-C bridge [67])
 - inotify [62] (through the Python pyinotify [63] module)
 - SQLite [68] (through the Python sqlite3 [73] module and the PHP PDO [108] database abstraction layer)
 - django-storages [89]
 - cssutils [83]
- Uses the following third party applications
 - ImageMagick [79]
 - pngcrush [80]
 - jpegtran [81]
 - gifsicle [82]
 - YUI Compressor [84]
- Supports the following storage systems
 - FTP (via django-storages, through the Python ftplib [96] module)
 - Amazon S3 [97] (via django-storages, through the Python boto [99] module)
 - Amazon CloudFront [98] (via django-storages, through the Python boto [99] module)
- Integrates with the following applications
 - Apache HTTP Server

12 Feedback from businesses

I had a nice list of seven companies who wanted to test my bachelor thesis, either the CDN integration for Drupal using the CDN or just the daemon (for either syncing files to a static file server or to back up servers).

I gave them more than three weeks time to test this, which is a fairly large amount of time, considering that the whole timespan of the thesis was about 4 months. Unfortunately, these are actual companies, each with their own schedules and clients to keep satisfied. So, as was to be expected, the feedback I *actually* got was far more sparse than I hoped for. However unfortunate, this is of course understandable.

Three CDNs promised me to provide me (and the businesses that were going to test my work) free CDN accounts for testing purposes. Two of them immediately gave me accounts: SimpleCDN [109] and Rambla [110]. Thanks! I ensured the daemon's FTP support was compatible with their FTP servers.

There are *two* companies that did give feedback though. Which is of course still much better than none at all.

The company SlideME [111] evaluated the daemon I wrote, which is by far the most important part of this thesis. They will be using it on a Drupal web site though, but have to adapt the daemon first to match their infrastructure (see later). My sincere thanks go to SlideME! What is interesting is that their company is all about Android [113], another open source project.

SlideME is an industry pioneer in Android content delivery, creating the first application manager for discovering, downloading and purchasing of Android applications directly to the device.

They are planning on using it with the following infrastructure:

1. MySQL database instance (with few slaves)
2. Dedicated "editor instance" (i.e. a single web server for the users that are allowed to add content) with Amazon S3 to keep Drupal files and upload new ones.
3. Multiple "viewer instances" for regular users on Amazon Elastic Compute Cloud (EC2, these are virtual server instances in the cloud) behind Elastic Load Balancing (which distributes incoming traffic across multiple Amazon EC2 instances).

Kay Webber, whom works for SlideME and is responsible for integrating my daemon with the infrastructure, made four suggestions:

1. On his servers with the CentOS operating system , only Python 2.4 was available, whereas, my daemon requires Python 2.5. He made the changes necessary to get it to work in Python 2.4. He did not manage to find a pyinotify package for his Linux distribution however, making `fsmonitor.py` fall back to the polling mechanism. However, not all of the Python packages I use are guaranteed to work with Python 2.4 (because Python 2.4 is quite old already, it was released in November 2004). On most servers, it is possible to install Python 2.5 with a single command.
2. The daemon is run on the editor instance, but the database of synced files needs to be accessed from the viewer instances as well. Therefore, he is working on a patch to also support storing the synced files database in a MySQL database (which supports replication) instead of just SQLite.
3. He would like to be able to use environment variables in the configuration file. He is also working on a patch for this.
4. He needs the ability to upload “private” files to Amazon S3 (this works using query string authentication, i.e. placing a combination of a public key and a signature in the query string).

In response, I asked him a couple of questions:

- Was the documentation sufficient?

Thesis and readme are both very clear. Now I regret that I did not finish my own bachelor degree ;) So the answer to your question is yes.

- Did you find the setup easy? Do you think the configuration file is self-explanatory?

The configuration file was mostly but not *always* self-explanatory. For example, I missed the connection between `<source name="drupal_files">` and `<rule for="drupal_files">` and named them differently at the beginning. But I have no suggestions for this case. The documentation on setup is clear.

- Since you have already been writing patches: what did you think of the code structure?

Clear and self-explanatory.

- How would you rate the performance? (Keeping in mind that discovering files is orders of magnitude slower due to the fallback to polling for detecting changes.) How many files are you syncing and what is their combined size?

I have not started performance tests yet, so I cannot rate it.

The second company I got feedback from, is WorkHabit [112]. WorkHabit is a company that aims to make scalable Drupal web sites easier.

WorkHabit is a leading provider of Social Software, Content Distribution, and Cloud Computing solutions that specializes in building sustainable businesses online.

Jonathan Lambert, the CEO, was very enthusiastic about the inotify support of the daemon and the lossless compression of images (by removing metadata and optimizing the way they are stored internally). He also really liked the idea to be able to pick from within the CDN integration module for Drupal which server files would be served to a visitor.

His company has a product which allows customers to use his “virtual CDN”. This “virtual CDN” then uses multiple *actual* CDNs. This implies that files must be synced to multiple servers and that is where my daemon could be very useful. That is also the use he is evaluating it for.

So, overall, the impression seems to be very good. The feature set seems to be strongly appreciated. The documentation in general, setting it up and the configuration file were strongly approved. The code structure even more. Neither had the time yet to do real-world tests.

The trend in the answers suggests that the code structure is sufficiently solid and that it conceptually makes sense; that setup and configuration are sufficiently straightforward; all of which hopefully is enough for this application to become a healthy open source project.

13 Test case: DriverPacks.net

As a back-up plan in case there would not be much feedback from companies (as turned out to be the case), I wanted to have a web site under my own control to use as a test case. That web site is DriverPacks.net [114] (see the figure below for a screenshot of its homepage). It is the web site of an open source project, with more than 100,000 visits per month and more than 700,000 pageviews per month, with traffic coming from all around the world. These fairly large numbers and the geographical spread of its visitors make it a good test case for measuring the effect of a CDN. See figure 31 for a map and details. Visitors come from 196 different countries, although the top three countries represent more than a quarter of the visitors and the top ten countries represent more than half of the visitors. Nevertheless, this is still a very geographically dispersed audience.

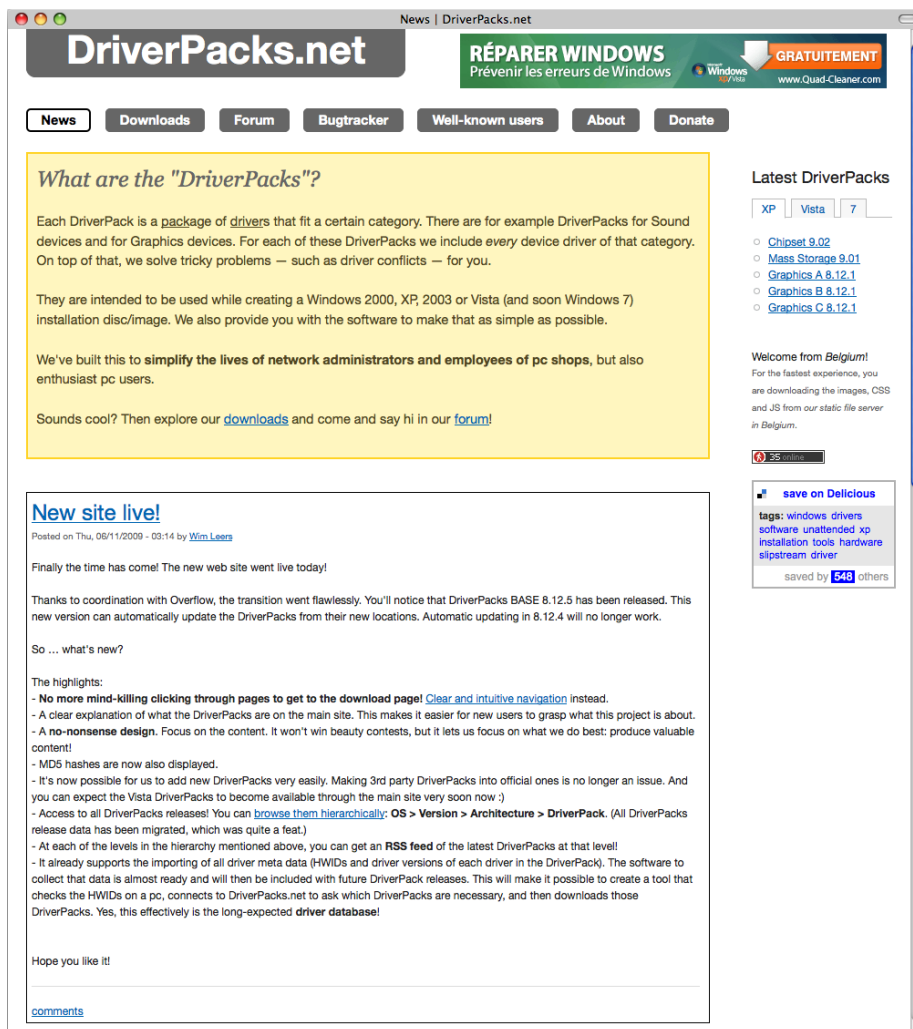


Figure 30: DriverPacks.net homepage.

The goal was obviously to port this site to Drupal (which is not a part of this thesis of course) and to install the Episodes module. During about a week, statistics would be collected while no CDN was being used. Then, I would install the daemon on the server to sync the files to a CDN. Next, I would install the Drupal CDN integration module. Then again for about a week, statistics would be collected while the CDN was being used. Hopefully, by visualizing the collected episode measurements, it would be confirmed that this had indeed had a positive effect.

Implementation

The web site was previously a combination of static HTML and a bit of PHP. On June 11, I launched the new web site, which was built using Drupal. Drupal's CSS and JavaScript aggregation options were enabled. Due to this, Drupal combines the CSS files for the same media type (`screen, all ...`) and in the same section (header or footer) and the JavaScript file in the same section (header or footer), thereby already significantly reducing the number of HTTP requests and page loading time. If I would have turned this on when enabling CDN integration, the effect would have been far more significant. But the comparison would have been unfair. Hence, these options were enabled, right from the beginning.

Also the JavaScript for the Google Analytics service has been cached on the web server since the launch of the web site, because the Google Analytics module for Drupal supports it. This makes it even harder to show the effect.

Finally, DriverPacks.net is a site with virtually no images (because it slows down the page loading, but mostly because I am not a designer). Only two images are loaded for the web site's style, one is loaded for each of the two statistics services and one is loaded for the ad at the top. Only the two images that are part of the web site's style can be cached. This makes it again more difficult to show the effect.

This suggests that the results that will be observed should be magnified significantly for more media-rich web sites, which is typically the case nowadays. However, I did not have such a web site around to test with. It is likely that if the effect is noticeable on this web site, it is dramatically noticeable on a media-rich web site.

On June 21 around 2 AM GMT+1, I enabled CDN integration for all users. To show the potential of the daemon in combination with the CDN integration module however, I implemented the `cdn_advanced_pick_server()` function that the Drupal CDN integration module calls when it is running in advanced mode and when that function exists. It really allows you to create any desired logic for routing users to specific CDNs or static file servers, as is demonstrated in listing 6.

Both a static file server and a CDN are being used, but not just for demonstrative purposes. It was decided to use two servers because the latency to SimpleCDN's servers is higher than acceptable in Europe (around 100 milliseconds or more). They claim to have servers in Amsterdam, but for some reason all requests from

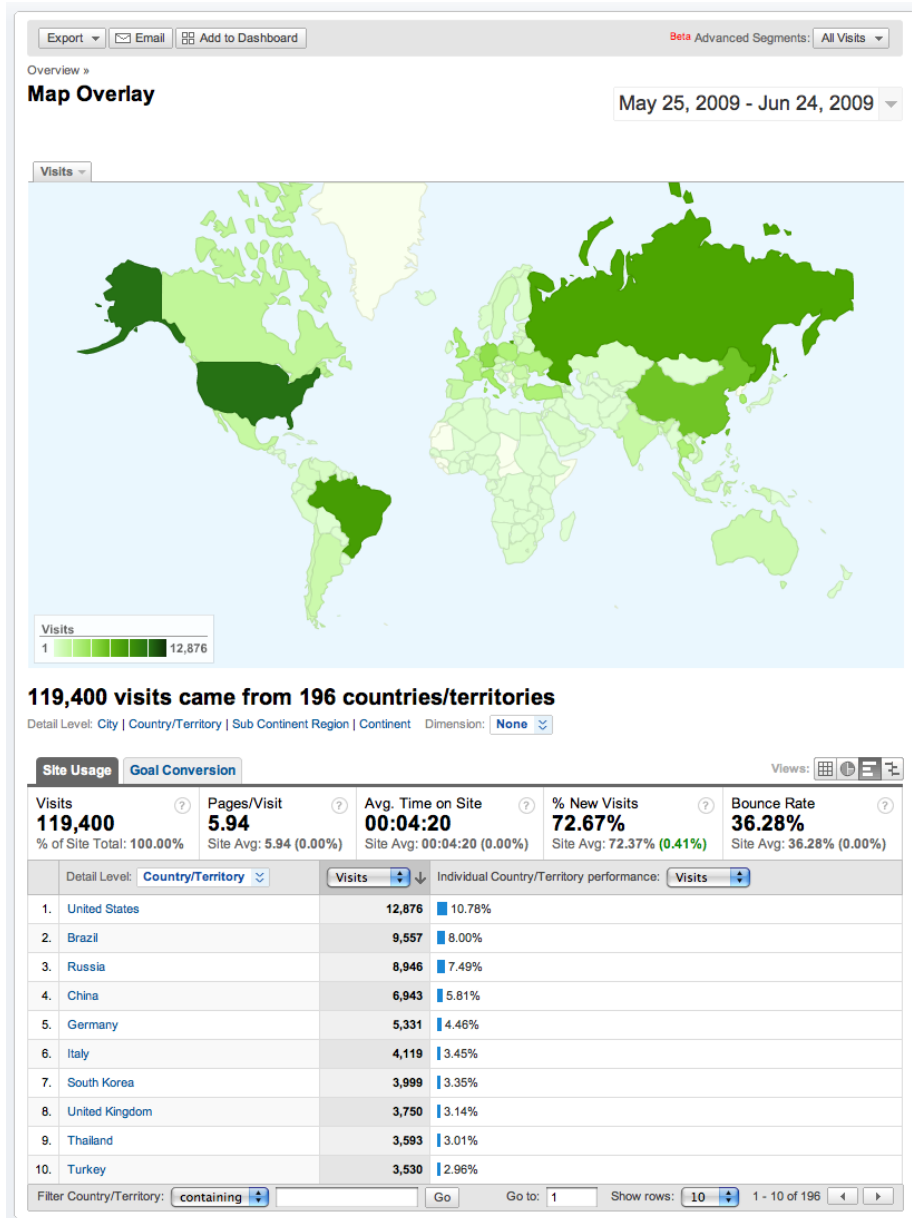


Figure 31: Google Analytics' Map Overlay view for DriverPacks.net.

Welcome from *Belgium!*
For the fastest experience, you are
downloading the images, CSS
and JS from our static file server in
Belgium.

Figure 32: Visitor location block.

my IP (in Belgium) are being routed to a datacenter in Phoenix, Arizona in the United States, which explains the high latency (this was discovered using the `traceroute` command). This was worse than *not* using a CDN for countries with low-latency connections to Belgium. Since latency is the major problem in the loading of CSS, JavaScript and other files referenced by the HTML (because of the multitude of round trips due to HTTP requests), I measured this using the `ping` command. I mostly relied on the “just ping” web service [115] to get ping times from all around the world, to get a global view.

If only SimpleCDN would have been used, the web site would have become significantly slower in Europe (also in Russia, but less dramatically) and it would have been very likely that no improvement could be observed from the collected statistics.

The static file server (labeled ‘warp’) server is located in Berchem, Belgium and uses the static.driverpacks.net domain name and is an Apache HTTP server instance (with far future Expires headers, gzip enabled and Last-Modified headers disabled) optimized for static file serving and the CDN (labeled ‘simplecdn’) server uses the cdn.driverpacks.net domain name. This is a DNS CNAME record pointing to e1h21.simplecdn.net (this server is for SimpleCDN’s “StormFront” service, which claims to have 9 POPs in the United States and 2 POPs in Europe). Files served from this domain have gzip enabled, far future Expires headers and no Last-Modified headers. There are other CDN services than SimpleCDN that provide POPs all over the world, as opposed to just two continents. They are typically far more expensive though.

The Drupal `ip2country` module [61] is used to map an IP to a country code. This module uses the database maintained by ARIN, the American Registry for Internet Numbers. ARIN is one of the five official Regional Internet Registries (RIR) responsible for assigning IP addresses. It is claimed to be 98% accurate, which should be sufficiently accurate. Via the `continents_api` module [117] (which I wrote as an addition to the `countries_api` module [116]), all country codes for Europe are collected in an array. Russia’s country code is appended to this array. This way, all European countries plus Russia are assigned to the ‘warp’ server. Visitors from all other countries are assigned to the ‘simplecdn’ server. This implies that files are being synced through the daemon to both the ‘warp’ server (using the symlink or copy transporter, see section 9.3.6) and to the ‘simplecdn’ server (using the FTP transporter, again see section 9.3.6).

Visitors are informed on the homepage from which server they are getting their static files, as can be seen in figure 32.

Listing 6: `cdn_advanced_pick_server()` function and helper function.

```
function driverpacksnet_map_country_to_server($country_code) {
    $countries_for_warp_server = continents_api_get_countries('EU') + array('RU');
    if (in_array($country_code, $countries_for_warp_server)) {
        return 'warp';
    }
    else {
        return 'simplecdn';
    }
}

/**
 * Implementation of cdn_advanced_pick_server().
 */
function cdn_advanced_pick_server($servers_for_file) {
    static $server_name;

    if (!isset($server_name)) {
        // Determine which server to use for the current visitor.
        $ip = $_SERVER['REMOTE_ADDR'];
        $country_code = ip2country_get_country($ip);
        $server_name = driverpacksnet_map_country_to_server($country_code);
    }

    // Try to pick the desired server - if the file being served is available on
    // our the desired server.
    foreach ($servers_for_file as $server_for_file) {
        if ($server_for_file['server'] == $server_name) {
            return $server_for_file;
        }
    }

    // In case our desired server does not have the file, pick the first server
    // that does have it.
    return $servers_for_file[0];
}
```

Episodes analysis - overall

2705623 episode measurements have been collected over 261724 page views (112127 of which also contain measurements of the back-end) from Thursday, June 11, 2009 to Thursday, June 25, 2009, with visitors coming from 168 countries.

Figure 33: Episodes analysis: overall.

One day later, again around 2 AM GMT+1, I added some optimizations: from that moment, Google Analytics' JavaScript code was being executed after the `window.onload` event. This caused the executing of JavaScript code and the loading of an invisible 1x1 image to be delayed until after the page was fully rendered, thereby speeding up the rendering of the page and thus the perceived page load time.

The CSS and JavaScript of the [delicious.com](#) bookmarking widget (see the bottom right of figure 30) on the homepage were also altered to be served from [DriverPacks.net](#)'s web server (and after syncing, these files were being served from either the static file server or the CDN). The AJAX request to [delicious.com](#)'s servers was being delayed until after the `window.onload` event, i.e. when the page was fully rendered. This is not annoying because the information in this widget is not critical.

Finally, I also modified the loading of the image necessary for the [whos.amung.us](#) statistics service [118]. Normally, this image is simply referenced from within the HTML. However, now it is being detected on the server side if the visitor has JavaScript enabled (Drupal then sets a `has_js` cookie, which can be detected on the server side) and in that case, a piece of JavaScript is inserted instead that is executed after the `window.onload` event, which inserts the HTML that loads the image. CSS is used to consume the whitespace until the image is loaded (where otherwise the image would have been), to prevent annoying relayouting (perceived as "flicker" by the end user).

Collected statistics

About 100 MB worth of statistics had been logged. These were then imported on June 25 (using the Episodes Server module, see section 8.4), resulting in a database table of 642.4 MB. More than 2.7 million episodes were collected over more than 260,000 page views. See figure 33 for details (note: in the figure, the times are in GMT, which explain the discrepancy in time span). All screenshots listed in this subsection are made from the Episodes Server module running on [DriverPacks.net](#).

This means episode measurements have been collected since the beginning of June 11 until the beginning of June 25, resulting in exactly two weeks of data, with the last 4 days having CDN integration. This is in fact too short to see the full effect of caching kicking in. [DriverPacks.net](#)'s visitors typically don't visit the site daily, but weekly or monthly. Hence many visitors have an empty cache, which drives the measured durations up. Fortunately, the effect of the CDN on visitors with an empty cache is clearly visible. The effect of visitors with primed caches is also visible, but less explicit.

Episodes analysis - page loading performance

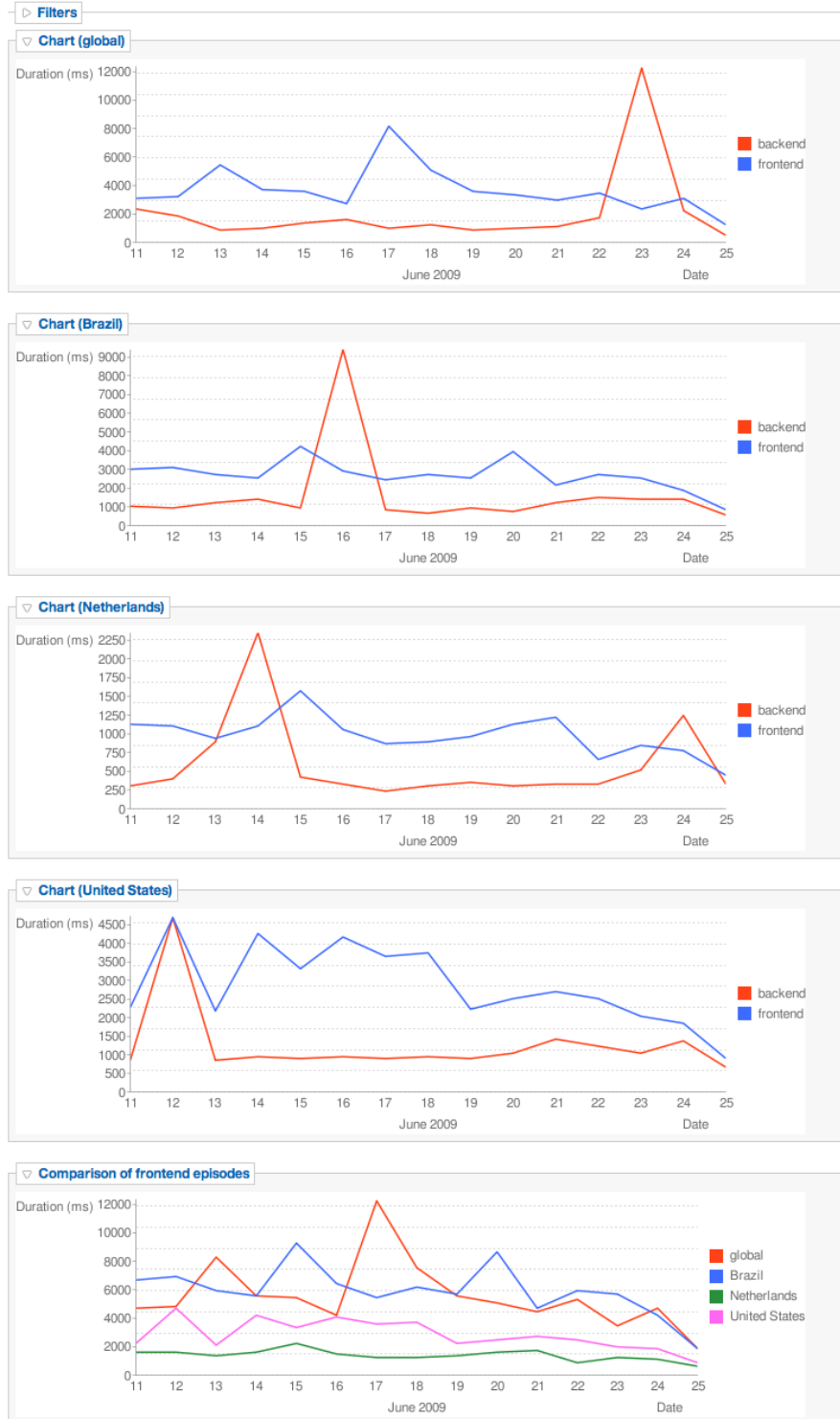


Figure 34: Episodes analysis: page loading performance.

Now the interesting part begins. There are five charts in figure 34. The first four display the average duration of the **backend** and **frontend** episodes per day, for a given territory of visitors. The first chart gives the results for all collected data, i.e. globally. The second chart is for Brazil, the third for the Netherlands and the fourth for the United States. As we have seen, the United States and Brazil represent the largest groups of visitors, so their results should closely match the average real-world performance there, which means they should be well represented in these charts. The Netherlands were chosen because a location very close to the 'warp' server was also needed. The Netherlands were chosen over Belgium because more visitors originate from there, making the results more representative. On average, the U.S. had about 2,800 page views per day, Brazil had about 1,500 page views per day and the Netherlands had about 350 daily page views.

The fifth and last chart compares the average duration of the **frontend** episodes per day of the three countries with the global averages.

As this bachelor thesis is not page rendering performance, but about page loading performance, the charted data for **backend** episodes can be ignored. I have no explanation for why it peaks at different times around the world.

The **frontend** episodes contain the durations of loading all CSS, JavaScript and images.

Looking at the **frontend** episode durations in the first four charts, it is clear that in the period from June 11 until June 21, the variability in page loading time was very large. This may suggest that measurements are not taken over enough page views from a global point of view, but in the case of the United States and Brazil, which both had more than 2,500 page views per day (which should be sufficient to cancel out anomalies), it suggests that the variability of network latency (due to distance or congestion) can be great. This can probably be explained by the fact that most files had to be retrieved from the web server in Belgium. Belgium is a long distance from Brazil and the United States and therefore there are more opportunities (at every hop along the route) for the latency to increase, which is possibly the reason for the large variability.

Starting June 21, which is the date when CDN integration was enabled, there is a slight increase in some countries, but globally, there is a slight decrease. This is to be expected, as the frequent visitors have to download the static files again, this time from the CDN ('simplecdn' server) or static file server ('warp' server). On June 22, when another batch of changes was applied and the cached CSS and JavaScript files were once again obsolete, there is a slight increase globally, but a significant drop in the Netherlands.

It is likely that this is because whereas Dutch visitors previously had to wait for as much as five round trips to the United States (three to the delicious.com servers, one to Google Analytics' server and one to the additional statistics service's server), that is now reduced to just three (the CSS and JavaScript files for the delicious.com widget are being served from the CDN or, in the case of the Netherlands, from the static file server in Belgium).

However, starting on June 23, there is a clear, worldwide, downward trend. Especially in Brazil and the United States, this trend is very prominently visible in the charts. It is less strong in the Netherlands, because they were getting

their files already from a server with a fairly small latency. The reason it drops even further for the Netherlands likely is that the static file server is configured properly to maximize client-side caching (i.e. in the visitor's browser), whereas before the CDN and static file server were being used, static files were being served using the default Apache settings.

Finally, there is the analysis of the duration of the episodes (see figure 35) themselves, regardless of country. Through this chart, it is possible to examine which episode have the greatest impact on the overall page loading time and would thus benefit the most from optimizations. Unfortunately, I did not have the time to create a chart that displays the evolution over time. So it is currently impossible to see how much a specific episode has improved over time.

However, it is very clear that the `header.js` episode (which represents the episode of loading the JavaScript in the `<head>` element of the HTML document) is by far the episode with the longest duration, thus is the best candidate for optimization. Probably even after CDN integration, this episode will remain the longest.

Conclusion

Globally, there has been a strong decrease in the page loading time and thus a strong increase in page loading performance. And since this test was conducted on a media-poor web site, it is likely that the effect is dramatically more noticeable on a media-rich web site. Since the downward trend lasted two days, it seems likely that the page loading time will become less variable because the static files have been cached in the browser.

Especially for countries that are located far from Belgium, such as Brazil and the United States, variability seems to have decreased significantly. This suggests that adding more servers around the world, at strategic locations (for example in the case of DriverPacks.net, Bangkok in Thailand is the city that generates the most visitors around the world according to Google Analytics), may be a very effective method for improving page loading performance. Depending on the web site, this strategy may achieve equal or better performance than a CDN with servers all around the world, at a fraction of the cost.

It is certain that every web site can benefit from an analysis like the one above. The strategy to optimize page loading performance is different for every web site.

Episodes analysis - episodes

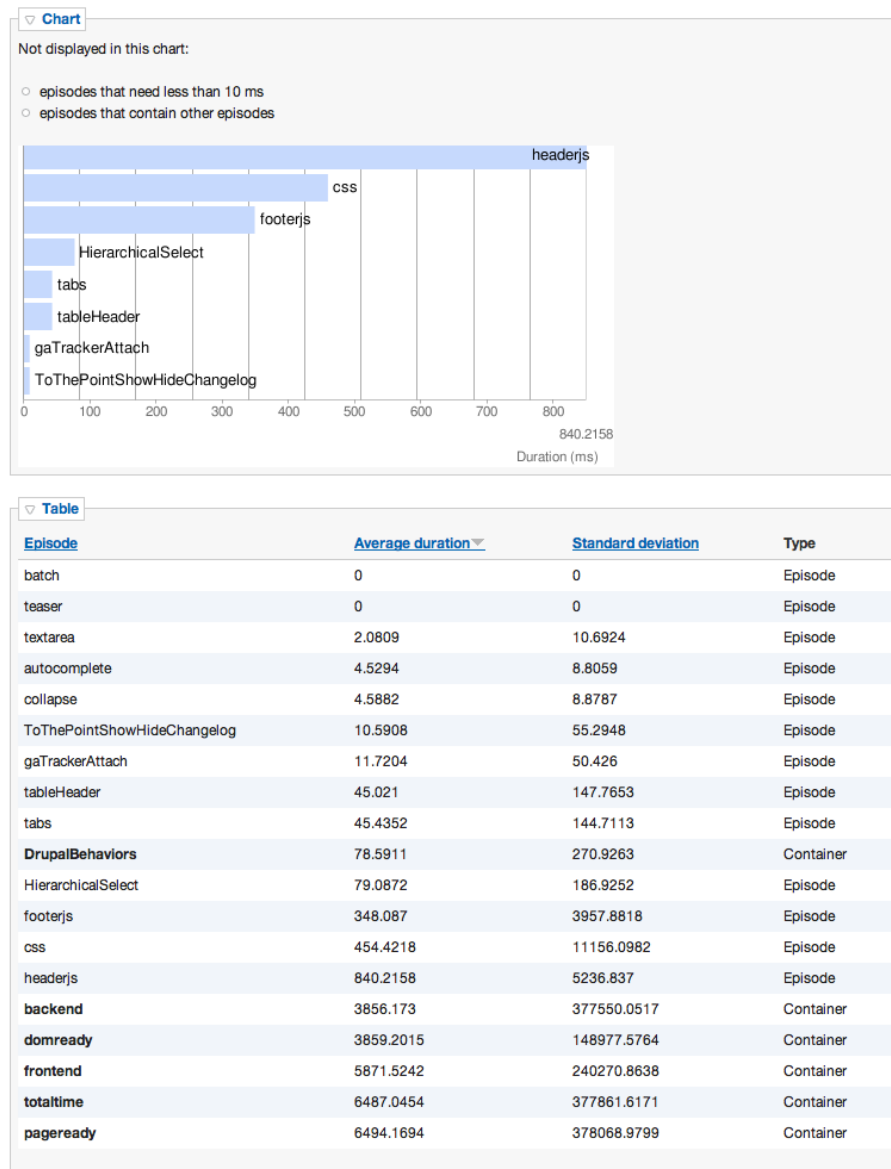


Figure 35: Episodes analysis: episodes.

14 Conclusion

The goal of this bachelor thesis is improving Drupal's page loading performance. But in order to guarantee that my attempted improvements to Drupal actually have an effect, I had to be able to measure the results.

After analyzing a broad set of page loading performance profiling tools (and related tools), it became clear that Episodes (which was written as a prototype by Steve Souders, whom can be considered the evangelizer of page loading performance) was the best candidate: it is the only one that is capable of measuring the real-world page loading performance, the results from the actual people using it. Furthermore, it has the most potential for becoming the standard profiling tool over the next few years and even to become integrated with future browsers.

So I polished the code, made it ready (or at least usable) for the real world and integrated it with Drupal: the Drupal Episodes module. The integration with Drupal happens in such a way that all Drupal behaviors (all JavaScript behaviors are defined using a standardized Drupal JavaScript API) are automatically measured. All that is needed, are a couple of changes to the theme to make the measurements cover all aspects.

This module is ready for use in production.

I also made another Drupal module (the Episodes Server module) that is able to import the logs collected through Episodes and visualize the measurements through charts. Thanks to these charts, you can track the real-world page loading performance over time and even compare the page loading performance of multiple countries simultaneously with the global page loading performance. It also allows you to find out which episodes are the biggest bottlenecks for a better page loading time.

This module is not ready for production, but is a good base to start from. The code that imports the logs into the database is covered by unit tests. The code is licensed under the GPL and available at <http://drupal.org/project/episodes>.

Then there is the daemon for syncing files, of course. This was the most important part of this thesis. As odd as it may seem, there seems to be nothing similar in existence, or at least not publicly available (not even commercially). If it did exist, I would surely have been informed by one of the tens of people who know about the concept of my bachelor thesis.

I started with the configuration file design. The configuration of the daemon happens through an XML file that is designed to be easy to use if you are familiar with the terminology. Several people who are familiar with the terminology were asked to look at it and immediately responded that it made sense. It is important that this is easy, because this is the interface to the daemon.

I decomposed the daemon into major components and started with the most simple one, because I was writing this in Python, a language that I had never used before, but was rumored to be extremely easy to program in — this ease was supposedly partially thanks to the availability of modules for nearly everything imaginable. Fortunately, that turned out to be the case, although I had

been fearing for a long time that I had to write all the code for transporters myself — a near impossible challenge given the timeframe.

This is why the daemon in fact is just a set of Python modules that can be used in any application. For example, in the case of the `fsmonitor.py` module — which abstracts file system monitors on different operating systems, thereby creating a cross-platform API — it is certainly possible that it will be reused by other applications.

Thus I wrote a fairly large set of Python modules: `config.py`, `daemon_thread_runner.py`, `filter.py`, `fsmonitor.py` (with one subclass for every supported operating system), `pathscanner.py`, `persistent_list.py`, `persistent_queue.py`, `processor.py` (with a set of subclasses, one for each processor) and `transporter.py` (with subclasses that are very thin wrappers around Django custom storage systems). Whenever feasible, I wrote unit tests. But because this application involves a lot of file I/O and network I/O due to its nature, this was often extremely hard and thus skipped. For `fsmonitor` support in Linux, I was able to build upon the `pyinotify` module and for the transporters, I was able to completely reuse Django's custom storage systems. All other modules are completely my own product.

This has an interesting side effect: `arbitrator.py`, the module that links all these stand-alone modules together into a single whole, i.e. which arbitrates between all different modules, can easily be refactored completely. While it is almost a thousand lines of code (albeit with a lot of comments), the code could easily be rewritten completely as part of a redesign, and you would only have to write the logic that links all modules together. This means that in a worst case scenario, for example, the daemon is found to have a bottleneck in a certain situation due to a design flaw, it can easily be adapted, because all the logic of the daemon itself is embedded in a single module. Only that part must be rewritten.

Because it is impossible to be sure that the daemon works correctly and reliably in every environment and with every possible configuration, it is recommended that a company first simulates its use case and verifies that the daemon works to its expectations. The code is licensed under the GPL and will be made available on a web site (but a name for the daemon has not yet been decided on). Hopefully, others will start contributing to it to improve it further to make it suitable for more situations.

A Drupal module for easy integration with CDNs was also written: the CDN integration module. However, I first had to write a patch for Drupal core, because it is necessary to be able to alter the URLs to files that are generated by default, because it is necessary to point them to the files on the CDN instead of those on the web server. A patch for Drupal 7 (which is currently in development), to make this functionality part of Drupal core in the future, has had very positive reviews, but must still go through the minutious peer review process and will likely soon get committed.

In the Drupal module, two modes are available: basic and advanced. In basic mode, it can only be used with Origin Pull CDNs. But because it now makes this very easy, whereas it previously required a lot of manual fiddling, this alone is a worthy addition to the Drupal contributed modules. However, in advanced mode, it becomes truly interesting: then it uses the synced files database generated by the daemon to rewrite file URLs. It is even possible to implement a

special callback function which can be used to select a specific server to serve the files from based on properties of the visitor (location, membership type, anything).

This module is also ready for use in production. The code is licensed under the GPL and is available at <http://drupal.org/project/cdn>.

The feedback from companies was underwhelming in numbers but overwhelming in positivity. I could not have hoped for more positive feedback. The possibilities the daemon offers were strongly appreciated. The code structure of the daemon was described as “clear and self-explanatory” and the documentation (of the daemon itself and my description of it in this thesis text) as “very clear”. It even made a reviewer regret that he did not finish his own bachelor degree. This reviewer was even so enthusiastic that he already started writing patches for the daemon so that it would better suit his infrastructure. This suggests that it is feasible that the daemon will become a lively open source project.

Finally, the results of my own testing case confirmed the thesis that integrating Drupal with a CDN could further improve the page loading performance. While the results (as logged via the Episodes module) were not as explicit as they could have been for a media-rich site (my test case was a media-poor web site), the difference was still clearly discernable in the charts (as generated by the Episodes Server module). Despite the fact that the web site had already been optimized with the mechanisms available in Drupal by default, integrating it with a CDN (via the CDN integration module and the daemon), clearly improved the overall, worldwide page loading performance.

References

- [1] *High Performance Web Sites*, Steve Souders, 2007, O'Reilly, <http://stevesouders.com/hpws/>
- [2] *Drupal*, <http://drupal.org/>
- [3] *History*, <http://drupal.org/node/769>
- [4] *Principles*, <http://drupal.org/principles>
- [5] *Drupal.org Explosion and Trends*, Steven Wittens, March 2007, <http://acko.net/blog/drupal-org-explosion-and-trends>
- [6] *Drupal 6 growth*, Dries Buytaert, April 2009, <http://buytaert.net/drupal-6-growth>
- [7] *Drupal sites*, Dries Buytaert, <http://buytaert.net/tag/drupal-sites>
- [8] *On backward compatibility: the drop is always moving*, <http://drupal.org/node/65922>
- [9] *Coding standards*, <http://drupal.org/coding-standards>
- [10] *Security Team*, <http://drupal.org/security-team>
- [11] *Modules*, <http://drupal.org/project/modules>
- [12] *Themes*, <http://drupal.org/project/themes>
- [13] *Design Fast Websites*, Nicole Sullivan, 2008, <http://www.slideshare.net/stubbornella/designing-fast-websites-presentation>
- [14] *We're all guinea pigs in Google's search experiment*, Stephen Shankland, http://news.cnet.com/8301-10784_3-9954972-7.html
- [15] *Usage statistics for Drupal*, <http://drupal.org/project/usage/drupal>
- [16] *Improving Drupal's page loading performance*, Wim Leers, January 2008, <http://wimleers.com/article/improving-drupals-page-loading-performance>
- [17] *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
- [18] *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
- [19] *Performance Analysis*, http://en.wikipedia.org/wiki/Performance_analysis
- [20] *UA Profiler*, Steve Souders, 2008, <http://stevesouders.com/ua/>
- [21] *Cuzillion*, Steve Souders, 2008, <http://stevesouders.com/cuzillion/>

- [22] *Cuzillion*, Steve Souders, 2008, <http://www.stevesouders.com/blog/2008/04/25/cuzillion/>
- [23] *Hammerhead*, Steve Souders, 2008, <http://stevesouders.com/hammerhead/>
- [24] *Hammerhead: moving performance testing upstream*, Steve Souders, September 2008, <http://www.stevesouders.com/blog/2008/09/30/hammerhead-moving-performance-testing-upstream/>
- [25] *Firebug*, <http://getfirebug.com/>
- [26] *Fasterfox*, <http://fasterfox.mozdev.org/>
- [27] *YSlow*, Steve Souders, 2007, <http://developer.yahoo.com/yslow/>
- [28] *Exceptional Performance*, 2007, <http://developer.yahoo.com/performance/index.html>
- [29] *Best Practices for Speeding Up Your Web Site*, 2008, <http://developer.yahoo.com/performance/rules.html>
- [30] *YSlow: Yahoo's Problems Are Not Your Problems*, Jeff Atwood, 2007, <http://www.codinghorror.com/blog/archives/000932.html>
- [31] *YSlow 2.0 early preview in China*, Yahoo! Developer Network, 2008, http://developer.yahoo.net/blog/archives/2008/12/yslow_20.html
- [32] *State of Performance 2008*, Steve Souders, 2008, <http://www.stevesouders.com/blog/2008/12/17/state-of-performance-2008/>
- [33] *Apache JMeter*, <http://jakarta.apache.org/jmeter/>
- [34] *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>
- [35] *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>
- [36] *Gomez*, <http://www.gomez.com/>
- [37] *Keynote*, <http://www.keynote.com/>
- [38] *WebMetrics*, <http://www.webmetrics.com/>
- [39] *Pingdom*, <http://pingdom.com/>
- [40] *AJAX*, <http://en.wikipedia.org/wiki/AJAX>
- [41] *Selenium*, <http://seleniumhq.org/>
- [42] *Keynote KITE*, <http://kite.keynote.com/>

- [43] *Gomez Script Recorder*, http://www.gomeznetworks.com/help/Gomezu/main/Gomez_university/3_Gomez_Script_Recorder/toc.htm
- [44] *WhitePages*, <http://whitepages.com/>
- [45] Velocity 2008, *Jiffy: Open Source Performance Measurement and Instrumentation*, Scott Ruthfield, 2008, <http://en.oreilly.com/velocity2008/public/schedule/detail/4404>
- [46] Velocity 2008, *video of the Jiffy presentation*, Scott Ruthfield, 2008, <http://blip.tv/file/1018527>
- [47] *Jiffy*, <http://code.google.com/p/jiffy-web/>
- [48] *Jiffy Firebug Extension*, <http://billwscott.com/jiffyext/>
- [49] *Episodes: a Framework for Measuring Web Page Load Times*, Steve Souders, July 2008, <http://stevesouders.com/episodes/paper.php>
- [50] *Episodes: a shared approach for timing web pages*, Steve Souders, 2008, <http://stevesouders.com/docs/episodes-tae-20080930.ppt>
- [51] *Google Analytics*, <http://google.com/analytics>
- [52] *Episodes*, Steve Souders, 2008, <http://stevesouders.com/episodes/>
- [53] *Episodes: a Framework for Measuring Web Page Load Times*, Steve Souders, July 2008, <http://stevesouders.com/episodes/paper.php>
- [54] *Episodes Drupal module*, Wim Leers, 2009, <http://drupal.org/project/episodes>
- [55] *Episodes Example*, Steve Souders, 2008, <http://stevesouders.com/episodes/example.php>
- [56] *Batch API*, Drupal 6, <http://api.drupal.org/api/group/batch/6>
- [57] *Forms API*, Drupal 6, http://api.drupal.org/api/group/form_api/6
- [58] *Hierarchical Select module*, Wim Leers, http://drupal.org/project/hierarchical_select
- [59] *Google Chart API*, <http://code.google.com/apis/chart/>
- [60] *Browser.php*, Chris Schuld, 2009, <http://chrisschuld.com/projects/browser-php-detecting-a-users-browser-from-php/>
- [61] *IP-based Determination of a Visitor's Country*, <http://drupal.org/project/ip2country>
- [62] *inotify*, <http://en.wikipedia.org/wiki/Inotify>
- [63] *pyinotify*, <http://pyinotify.sourceforge.net/>
- [64] *FSEvents Programming Guide*, 2008, http://developer.apple.com/documentation/Darwin/Conceptual/FSEvents_ProgGuide/Introduction/Introduction.html

- [65] *FSEvents review*, John Siracusa, 2007, <http://arstechnica.com/apple/reviews/2007/10/mac-os-x-10-5.ars/7>
- [66] *Watch a Directory for Changes*, Tim Golden, http://tingolden.me.uk/python/win32_how_do_i/watch_directory_for_changes.html
- [67] *PyObjC*, <http://pyobjc.sourceforge.net/>
- [68] *SQLite*, <http://www.sqlite.org/>
- [69] *How SQLite is Tested*, <http://www.sqlite.org/testing.html>
- [70] *Appropriate Uses For SQLite*, <http://www.sqlite.org/whentouse.html>
- [71] *Well-Known Users of SQLite*, <http://www.sqlite.org/famous.html>
- [72] *shelve - Python object persistence*, <http://docs.python.org/library/shelve.html>
- [73] *pysqlite*, <http://docs.python.org/library/sqlite3.html>
- [74] *smush.it*, <http://smush.it/>
- [75] *Image Optimization Part 1: The Importance of Images*, Stoyan Stefanov, 2008, <http://yuiblog.com/blog/2008/10/29/imageopt-1/>
- [76] *Image Optimization Part 2: Selecting the Right File Format*, Stoyan Stefanov, 2008, <http://yuiblog.com/blog/2008/11/04/imageopt-2/>
- [77] *Image Optimization, Part 3: Four Steps to File Size Reduction*, Stoyan Stefanov, 2008, <http://yuiblog.com/blog/2008/11/14/imageopt-3/>
- [78] *Image Optimization, Part 4: Progressive JPEG ... Hot or Not?*, Stoyan Stefanov, 2008, <http://yuiblog.com/blog/2008/12/05/imageopt-4/>
- [79] *ImageMagick*, <http://imagemagick.org/>
- [80] *pngcrush*, <http://pmt.sourceforge.net/pngcrush/>
- [81] *jpegtran*, <http://jpegclub.org/>
- [82] *gifsicle*, <http://www.lcdf.org/gifsicle/>
- [83] *cssutils*, <http://cthedot.de/cssutils/>
- [84] *YUI Compressor*, <http://www.julienlecomte.net/blog/2007/08/11/>
- [85] *Rhino*, <http://www.mozilla.org/rhino/>
- [86] *JSMIn, The JavaScript Minifier*, Douglas Crockford, 2003, <http://javascript.crockford.com/jsmin.html>
- [87] *Django*, <http://www.djangoproject.com/>
- [88] *Writing a custom storage system*, Django 1.0 documentation, <http://docs.djangoproject.com/en/1.0/howto/custom-file-storage/>

- [89] *django-storages*, David Larlet et al., <http://code.welldev.org/django-storages/wiki/Home>
- [90] *Cloud Files*, http://www.rackspacecloud.com/cloud_hosting_products/files
- [91] *Support for CloudFiles CDN?*, Tomas J. Fulopp, 2009 <http://drupal.org/node/469526>
- [92] *django-cumulus*, Rich Leland, 2009 <http://github.com/richleland/django-cumulus/tree/master>
- [93] *FTPStorage: saving large files + more robust exists()*, Wim Leers, 2009, <http://code.welldev.org/django-storages/issue/4/ftpstorage-saving-large-files-+-more-robust>
- [94] *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>
- [95] *SymlinkOrCopyStorage: new custom storage system*, Wim Leers, 2009, <http://code.welldev.org/django-storages/issue/6/symlinkorcopystorage-new-custom-storage>
- [96] *ftplib — FTP protocol client*, <http://docs.python.org/library/ftplib.html>
- [97] *Amazon S3*, <http://aws.amazon.com/s3/>
- [98] *Amazon CloudFront*, <http://aws.amazon.com/cloudfront/>
- [99] *boto*, <http://code.google.com/p/boto/>
- [100] *MogileFS*, <http://www.danga.com/mogilefs/>
- [101] *Apache CouchDB*, <http://couchdb.apache.org/>
- [102] *Stopping and Restarting - Apache HTTP Server*, <http://httpd.apache.org/docs/2.2/stopping.html>
- [103] *Pipes and Filters*, http://en.wikipedia.org/wiki/Pipes_and_filters
- [104] *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>
- [105] *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>
- [106] *Pipes and Filters*, Enterprise Integration Patterns, <http://www.eaipatterns.com/PipesAndFilters.html>
- [107] *Currying*, <http://en.wikipedia.org/wiki/Currying>

- [108] *PDO*, <http://php.net/pdo>
- [109] *SimpleCDN*, <http://www.simplecdn.com/>
- [110] *Rambla*, <http://rambla.be/>
- [111] *SlideME LLC*, <http://slideme.org/>
- [112] *WorkHabit*, <http://workhabit.com/>
- [113] *Android*, <http://www.android.com/>
- [114] *DriverPacks.net*, <http://driverpacks.net/>
- [115] *Just Ping*, <http://just-ping.com/>
- [116] *Country codes API*, http://drupal.org/project/countries_api
- [117] *Continents API*, <http://drupal.org/node/255215#comment-1722758>
- [118] *whos.amung.us*, whos.amung.us