niteco
Powering Your Ambition

info@niteco.com
niteco.com

E-GUIDE / part 1 of 3

Improving Web Performance in Episerver

June 2018

How to Improve Web Performance in Episerver

When it comes to the appeal of a website, nothing beats good performance. You can have the fanciest presentation with great visuals, perfect structure, and amazing content – if your site takes too long to load, nobody will stick around to see the fruits of your hard work.

Performance is key, and as an Episerver MVP, I know how to tease all the power out of an Episerver website. In this three-part E-Guide, I will show you how to do it. I will explain simple, common-sense approaches to speeding up your site, tell you where your site can lose considerable weight, and reveal the little secrets of Episerver's platform.

EMVP - Linus Ekström, Chief Technology Officer
The 3rd developer of the original team that created Episerver, Linus is a prominent figure amongst the industry's professionals. His greatest strength lies in the combination of technical knowledge and an eye for the big picture.

Why Web Performance Matters

Your website's performance can make or break your business. This may sound harsh, but it's a fact. In 2018, tolerance for slow web performance among users is at an all-time low and this trend will continue as more online transactions are going mobile.

The Kissmetrics blog gives us the bare and sobering numbers: "A one-second delay in page response can result in a 7% reduction in conversions. If an e-commerce site is making $100,000 per day, this could potentially mean a loss of sales of $2.5 million every year." The longer a site takes to load, the more visitors leave for a more responsive one.

According to a survey conducted by the blog, 47% of consumers expect a site to load in two seconds or less. Mobile users are a little more forgiving, with one third of users saying they would wait no longer than ten seconds for a page to load.

With average page size growing from 929 KB in 2011 to 3034 KB in 2017 and images and videos responsible for most of this growth, good performance optimization is now more important than ever. Consumers are getting used to a richer and more compelling experience with nice images and, increasingly, video, all the while still expecting to be able to interact with a site instantly.

Causes of Bad Performance

The speed of a site results from a combination of a number of factors. The main causes of slow performance are:

- Latency
- Server response time
- Parsing of HTML and CSS
- Downloading required assets
- Parsing and execution of JavaScript
- Rendering

Not only can problems in these individual areas cause delays and slow down performance, they can actually combine their effects by getting in each other's way and increasing load times.

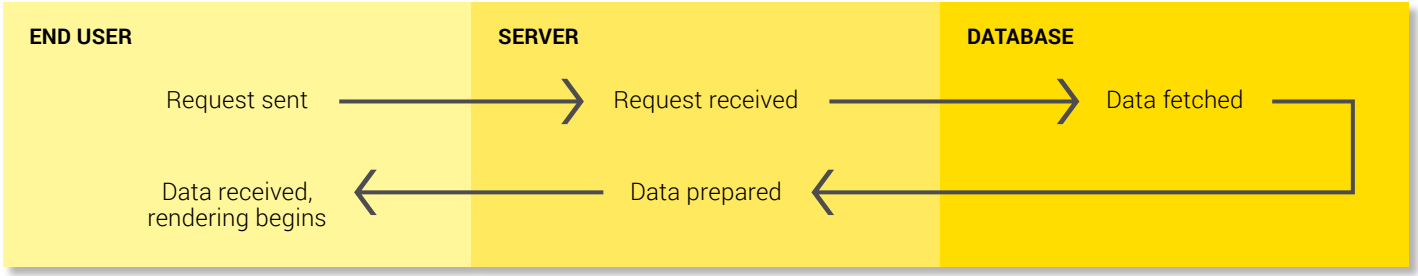
How Does a Page Request Work?

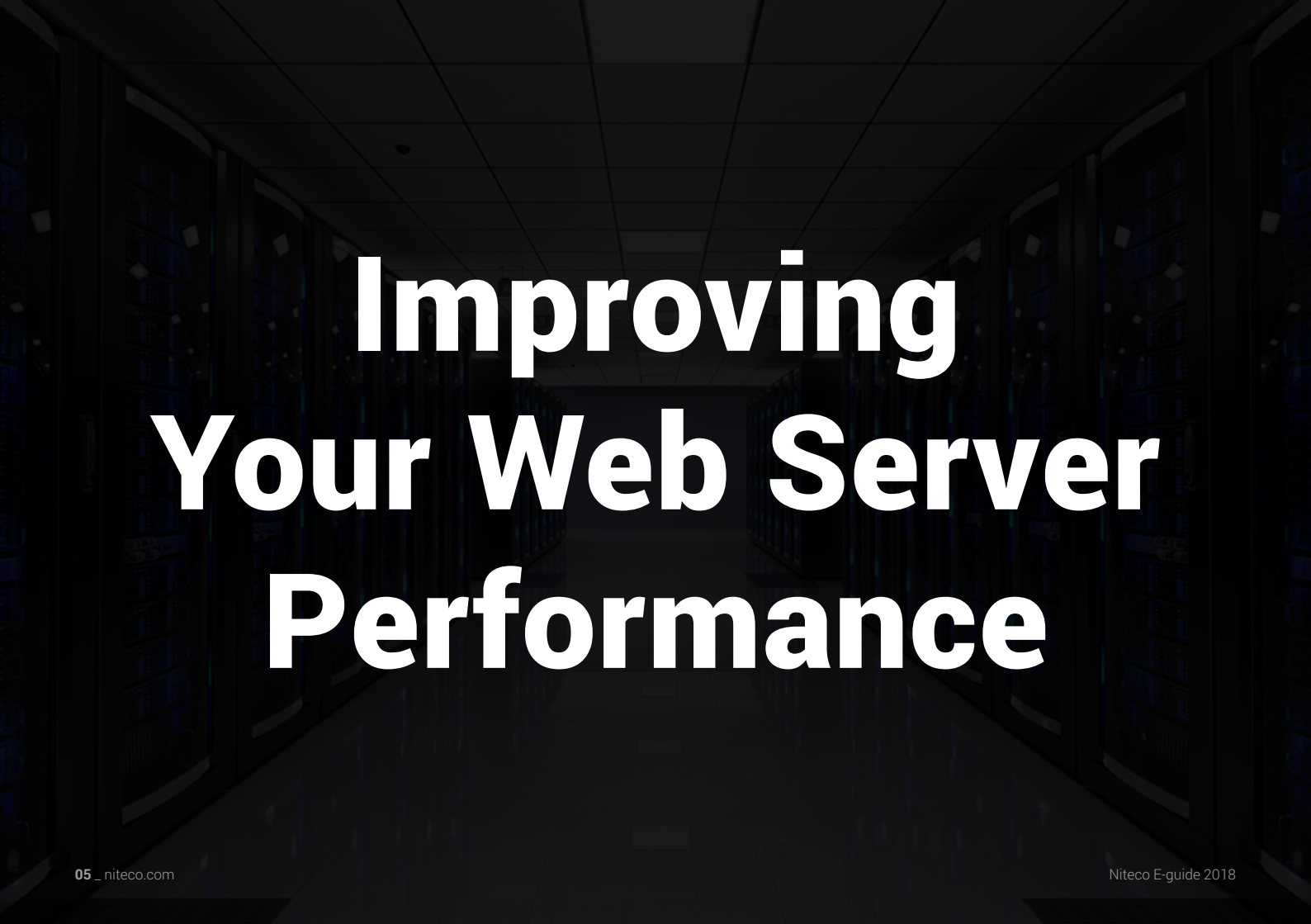
To understand what problems can occur between yourself and your visitor, it's important to know what actually happens when a user visits your site.

When a user enters your URL in their browser or clicks a link to your site, a request is sent from their device to your web server. If you do not already know the IP address of the server, a DNS lookup has to be made before you can make the request, to get the IP address of the web server. The server interprets the request and proceeds to collect all the necessary data from the cache or, in the worst case, the database. Database lookups are relatively slow, so a system like Episerver tries to cache as much of the data as possible to reduce the amount of database roundtrips. For an Episerver CMS solution with mostly well visited pages, there should be very little database traffic after the sites have been warmed up and been able to cache the data. For Commerce solutions or CMS solutions with lots of content that might not be that well visited, requests to the database might be more frequent.

Having retrieved this data, the server sends it to your device in order to display the requested page in your browser.

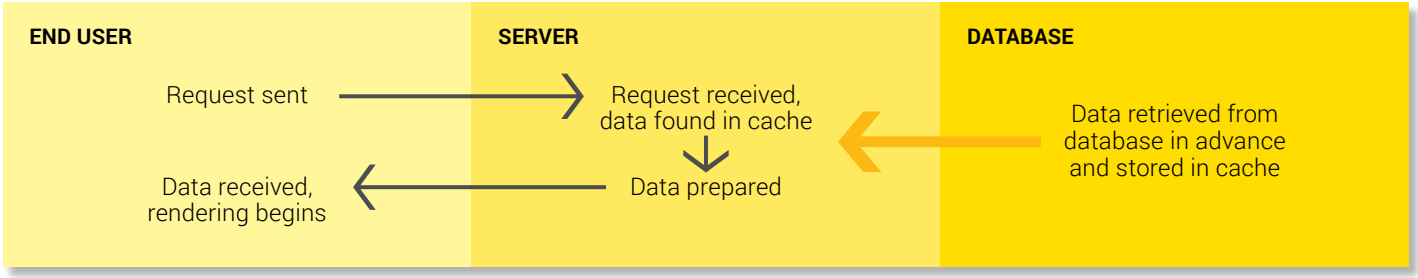
Your request accordingly undergoes four journeys, all of which have their own potential for latency and connection bottlenecks. If the user is located far away from your nearest web server, maybe even on the other side of the world, latency only gets worse.





Improving Your Web Server Performance

Content Caching



Caching is the temporary storage of content, done in order to speed up future loading speeds. Theoretically, a request for data made by the very first visitor to your website would have to go all the way to your site's database and retrieve what you want. As the data is pulled out of the database, it is delivered to your visitor and stored in different cache layers at the same time. When the next visitor makes that same request, the data can be delivered to him from one of the cache layers, without the need to go to the database. This saves time in loading the page. Content caching is done on a per-object basis. This means that even when a user visits a page that they have not yet visited, objects that have been used by other pages might still be taken from the cache, reducing the amount of calls to the database.

Caching can occur in several places. It can be done per object in your web application, as output cache with pre-generated HTML for parts of the page or the entire page, in a hypothetical CDN, and in your visitor's browser.

Each asset should have a Cache-Control header, which says how long the browser can cache the file locally. This makes subsequent visits to your site much quicker for that user, since the browser doesn't have to fetch those assets again. Assets that don't change frequently should be set with a cache time of at least one year. One common pattern for JavaScript that might change when a new deployment to the site is made is to add a dynamic part to the URL, for instance by adding a query string parameter. This way, you can configure aggressive caching while still being able to have the browser downloading a new version as soon as a new deployment is made.

What Is a CDN and Why Should You Use One?

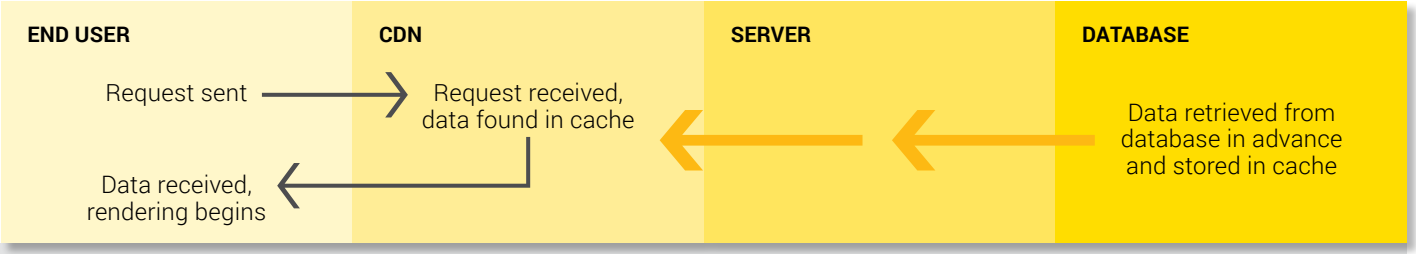
CDN stands for Content Delivery Network. If the solution has a CDN in place, this works as a proxy between the user and the web servers. A CDN is comprised of a multitude of nodes, usually placed on the internet backbone. When a DNS lookup is done on the site, the CDN closest to the user is returned. All traffic between the user and the site then effectively goes through the CDN. While this results in a slight overhead since the traffic needs to pass the CDN, the benefit is that the CDN can cache a lot of the assets for the site, for instance images, video, and scripts.

This means that visitors to your site retrieve the necessary data for loading your page from a server close to them instead of having to go to your own web server, which may be geographically located thousands of kilometers away. Reduced distance means reduced latency, decreasing load times.

By default, Episerver DXC comes with Cloudflare as its CDN provider. Setting up a Cloudflare service is simple and fast and can dramatically improve your website's performance, especially in areas further away from your web server's location, or when your site features a lot of images or video.

Static scripts and images can be stored in the CDN, leaving connections between the CDN server and your own system open. Since a CDN acts as proxy in between the user and web servers, it is very easy to add to existing solutions without much hassle.

CDNs today are more advanced than they were five years ago and can even help you with things like optimizing images. See "Image optimization" in the next edition of this E-Guide for more.



For an in-depth description of how a CDN works, I recommend this guide: ***https://www.incapsula.com/cdn-guide/what-is-cdn-how-it-works.html***

Your Website Is Your Castle

I like to imagine my sites as a castle, which has many lines of defense. Essentially, the risk is greater the further in the enemy gets. Ideally, you don't want the enemy to even get near your castle and instead be halted by the different levels of defense. Similarly, you would want your users to get what they need in their own browser's cache. However, few will be repelled so easily.

Make sure that static assets are cached on the client so they will not even be requested by the client at all if it has previously visited the site. Let´s call this border control. Anyone who passes the border will reach at least as far as your castle's moat, your CDN. This will stop quite a few attackers in their tracks, as you hopefully have all your most important assets cached in your CDN. This ensures that new visitors to the site can make use of requests made by other visitors in the same geographical region that have previously used the same CDN node.

If the warriors begin to scale your walls, they have reached your output cache, with which you can send a pre-processed copy of the requested page, thus protecting the inner layers. Output caching is basically storing the HTML for a request so that future visitors can use this without the server having to re-generate it.

The few that breach your gate and enter your courtyard, facing your few remaining guards, stand at your last line of defense, your server cache. This ensures that we can use content that has been fetched from the database over and over again, usually until an editor updates the item or the application is restarted.

If your last valiant fighters have been defeated, i.e. your server cache doesn't have the assets needed to fulfill the request, your inner keep stands defenseless: your database. Even though databases are increasingly fast, it´s hard to scale the database, and you want to be sure to avoid as many requests as possible down to this inner layer.

What I like about this analogy is that while the outer layers are large and should be able to manage a lot of requests, there will always be some that need to make their way down to the database, but if you are able to handle most requests in each layer, there should be fewer and fewer requests for each of them. This is optimal, since the outer layers are better at handling higher traffic than the inner layers.

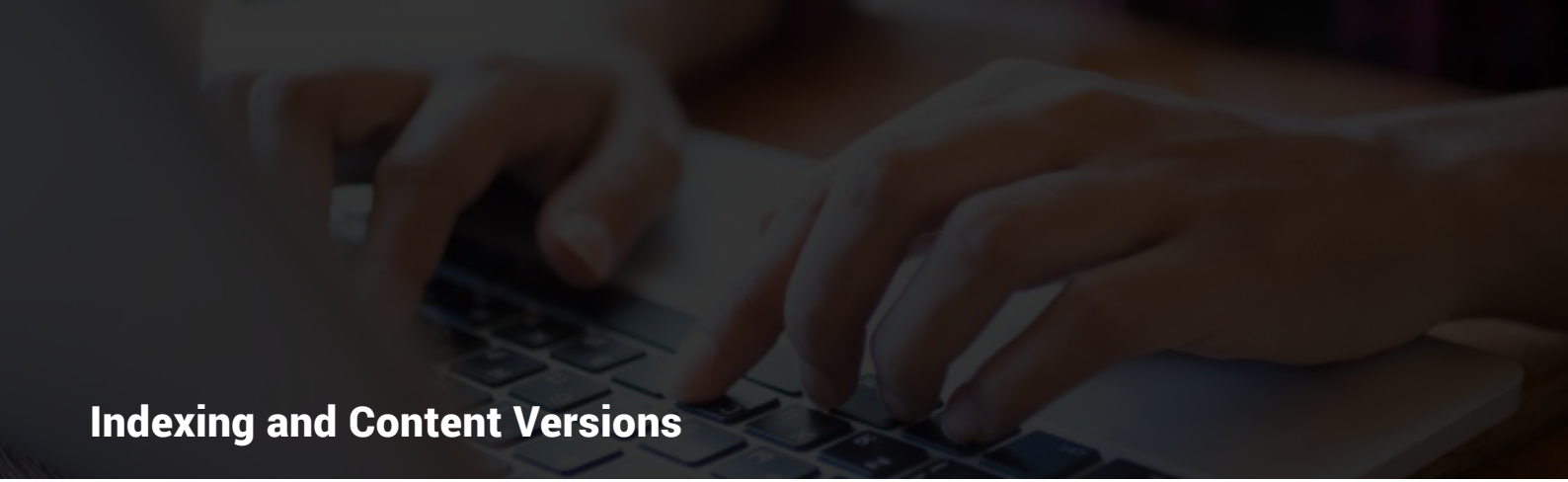
Donut Caching

Sometimes, caching an entire page is simply not the best way to go about business, especially when you engage in some levels of personalization or use other dynamic content like personal profile information.

To minimize the amount of data to be collected from the database, you can choose to go for a procedure called substitution caching or donut caching. With this approach, the parts of a page that can remain static are put in the output cache. Smaller parts of the page that need to accommodate dynamic content – the donut holes in this metaphor – are generated on the server, individually for each visitor.

The complementary approach is called donut hole caching. This means that a small part of a page is static – e.g. a particularly resource-intensive segment of the page - and can be cached, while the majority of the content is kept dynamic.

Donut caching that is done on the server has one big caveat: You cannot let the CDN cache the page, since it will always return the same HTML to all users. However, there are now CDN providers that enable you to tweak the output with JavaScript. Another approach could be to fill the holes with JavaScript on the client. This is a common approach when using Episerver Perform – personalized product recommendations.



Indexing and Content Versions

Even if you do everything to keep calls from reaching the database in order to improve performance, some inevitably have to actually go there. When that happens, you have to ensure that things go as smoothly as possible.

This means proper indexing, making it easier for the search process to find exactly what you are looking for in the database. It requires proper tagging and naming procedures as well as clean file structures.

It should be mentioned that while Episerver is very good at caching the published version of a content item, loading specific versions of an item will be done directly from the database and the content will be loaded from the database tables **tblWorkContent** and **tblWorkContentProperty**.

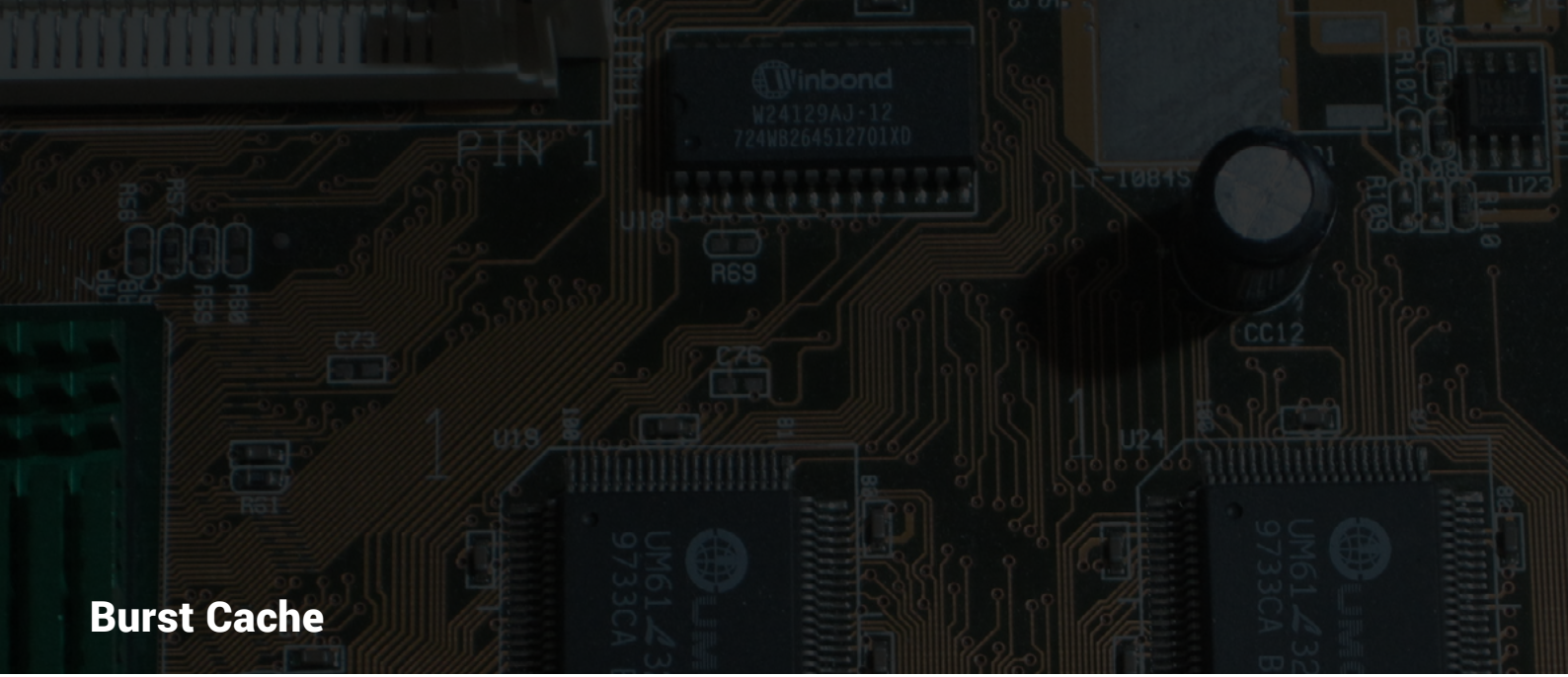
While keeping some old versions might be good or even a requirement for traceability, this comes with a cost. Specifically, the editorial environment might suffer if you have a lot of versions, since the data is always fetched from the database, from the versioned content tables.

We suggest that you keep no more than 10 versions of any given content item. This will help your database calls go smoothly and quickly. However, consider adjusting this number if you offer content in many different languages, as this can lead to versions stacking up again. A setting of 10 versions for a setup with five languages would mean that a maximum of 50 versions could theoretically be kept. This setting can be defined in the administrative interface or your configuration files.

Cache Invalidation

As mentioned before, Episerver is very good at keeping content objects cached. However, when a new version of an item is published, Episerver will invalidate the cache for this item for all server nodes. All servers then have to go to the database to fetch the new version of said item, though this is done only when the item is needed by the application. If there are several request being made in parallel before the data has been successfully loaded from the database, all requests waiting for this content are put on hold until the content has been loaded from the database. As long as the content is relatively fast to load, this should not be a problem. But if you have lots of traffic and evict content that is needed for many requests, this might mean that you fill up the request queue while waiting for the updated content to load.

This can cause a serious chokepoint to be created when all servers fetch the new version at the same time, specifically if there are a bunch of items being evicted from the cache at once and if it's content that is likely to be included in a lot of client requests. To labor the castle analogy once more, you are essentially lowering your bridges and opening all your gates, allowing your inner keep to remain defenseless. If you have a lot of content being updated frequently, you might want to consider a strategy to combat this, like the cache burst strategy explained below, or at least making sure that updates of content from external systems, like a PIM, are done outside the hours when the site has the most traffic.



Burst Cache

If a content update is not too time-sensitive, e.g. if it can wait until new data has been loaded, a burst cache update could be the way to go for you. When an item is evicted from the cache, the new version of a page is fetched from the database, and the old version remains available in the cache until the new content is loaded.

This means that the user never hits a page that is in the process of being rearranged or would need more time to load.

As soon as the updated version of the page is fully available in the cache, the old version is deleted and replaced.

Users can then retrieve this new version from the cache without having to wait for their request to get answered by the database. Episerver does not support Burst Cache by default, but there are several Episerver implementations I know about that use this strategy, since Episerver supports using standard .NET object caching and eviction strategies.

Image Optimization

Images account for a major share of data needed to display a web page. They are what makes sites vibrant, informative, and fun. They are also what turns the kilobytes needed to load a page into a serious number of megabytes. Therefore, optimizing the images used for your site is imperative, regardless of whether you are using the above-mentioned lazy load pattern.

When doing image optimization, it's obviously most important to keep file sizes small. However, you also need to offer a certain quality of image if you don't want your site to look cheap. Most image optimization software allows configuration of the quality of the optimized images.

Is a visitor coming from a smartphone? You can use a smaller picture. Are they on a laptop? You're going to need something bigger with good resolution. Are they on a 4K monitor? Be ready to present them with a crisp image to show that you know what you are doing. It´s quite common to configure at least three different image sizes on the site.

Retina displays have their own requirements, meaning you might want to use an extra version of every image to be shown on a retina device. Using this approach instead of using one file for all devices that visit your site will help you keep down the amount of data that has to be delivered on average.

A commonly used library for .NET-based solutions for resizing images for different devices is ImageResizer.net. There are two different pattern on how to resize images: when an image has been uploaded or when a user first requests the image with a given device size. While we will not take a stand for any one of these different approaches, we can give some guidance on the pros and cons:

Resize on upload

Pros:

- There is no delay when the first user requests the image and size.
- Can be done on a separate server to offload effect on the frontend servers.

Cons:

- All versions of an image are created when first uploaded. While this could (and should) be done asynchronously to not affect the application and potentially the user interface, this can have some negative effects on server performance, especially when importing a lot of new images, for instance through a scheduled job.

Image Optimization

Resize on demand

Pros:

- Can delay resizing of images to when they are actually needed. Some images might never be used externally.

Cons:

- The first user loading an image with a given size will suffer a slight delay when loading the image, since it has to be generated.

See **http://imageresizing.net/** for more information. One of the Episerver Most Valued Professionals, Valdis Iljuconoks, has actually created an Image Resizer plug-in that makes it possible to store the generated image variants in a blob provider, along with the original image: **https://github.com/valdisiljuconoks/ImageResizer.Plugins.EPiServerBlobReader**

Another important thing to know is that certain image file types need less storage space than others while seeming no different to the human eye. Take these examples:

PNG – Size 317KB

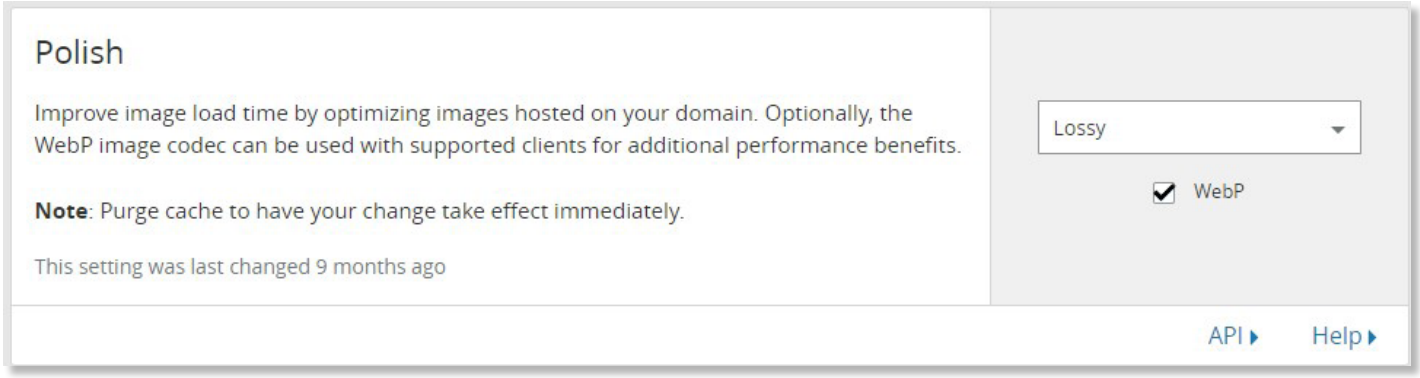
JPG – Size 65KB

WebP – Size 35KB

Using Modern Image Formats

Though older file formats like JPEG and GIF are still common on the web, there are newer file variants that make it possible to reduce file size while still maintaining the same quality, through better compression algorithms. WEBP, JPEG XR, and JPEG 2000 are examples of such formats. Compare the images above to see the differences in the images. The caveat is that not all browsers support these formats, so adding support for this on the server means additional work if you still want to support some older mainstream browsers, like IE 11.

If you are running Episerver DXC, it comes with Cloudflare CDN, which has a built-in image optimization function called Polish. This is a really neat feature that enabls the swapping of image formats for browsers that support the format WEBP. When an image is loaded to the CDN, it will investigate if it can save in file size by converting the image to WEBP. If it can, subsequent requests to the image done by browsers that support WEBP will get the file delivered as WEBP. The neat thing here is that the file name will still be the same, for instance woman.jpg, but the



type of the file will be changed in the header of the response. This means that this can be turned on without making any changes whatsoever to the application. Turning this on is as simple as checking a checkbox if you have access to the Cloudflare admin UI.

This does not create different image versions for different sizes, but can reduce the size of images. By default, Episerver will configure the quality setting for image optimization to 85. This can be changed by opening a ticket with Episerver's hosting. If you want to read more about Polish, you can do so here

https://blog.cloudflare.com/introducing-polish-automatic-image-optimizati/

See You Next Time!

Thank you for reading Part 1 of 3 of our E-Guide. You will be able to download Part 2 on July 5th.

In the next part of the series, I will tell you about the benefits of HTTP/2, how you can make sure scripts are executed smoothly and don't gum up the works, and why you should consistently test the performance of your site.

POWERING YOUR EPISERVER AMBITION

Niteco AB _ **STOCKHOLM**
Norrtullsgatan 6, 5tr, SE-113 29
Stockholm, Sweden
+46 (0) 700 355 830 | sweden.info@niteco.se

Niteco Group Ltd. _ **SYDNEY**
PO Box 868 Rozelle NSW
Australia 2039
+61 (0) 405 208 629 | australia.info@niteco.com

Niteco Group Ltd. _ **HONG KONG**
36/F, Tower Two Times Square
1 Matheson Street, Causeway Bay, Hong Kong
+84 (0) 128 801 2674 | hk.info@niteco.com

Niteco Vietnam Co. Ltd. _ **HO CHI MINH CITY**
E.Town Building 1, 2nd Floor, 364 Cong Hoa Street
Ward 13, Tan Binh District, HCM City, Vietnam
+84 (0) 286 297 1215 | info@niteco.com

LONDON _ Niteco Group Ltd.
3 More London Riverside, London, SE1 2RE
United Kingdom
+44 (0) 746 012 2355 | uk.info@niteco.co.uk

SAN FRANCISCO _ Niteco Group Ltd.
38505 Bautista Canyon Way, Palm Desert
CA 92260, USA
+1 (0) 415 871 2455 | usa.info@niteco.com

HANOI _ Niteco Vietnam Co. Ltd.
C'Land Tower, 14th Floor, 156 Xa Dan II Street
Dong Da District, Hanoi, Vietnam
+84 (0) 243 573 9623 | info@niteco.com

niteco.com

Niteco E-guide / NITECO1806