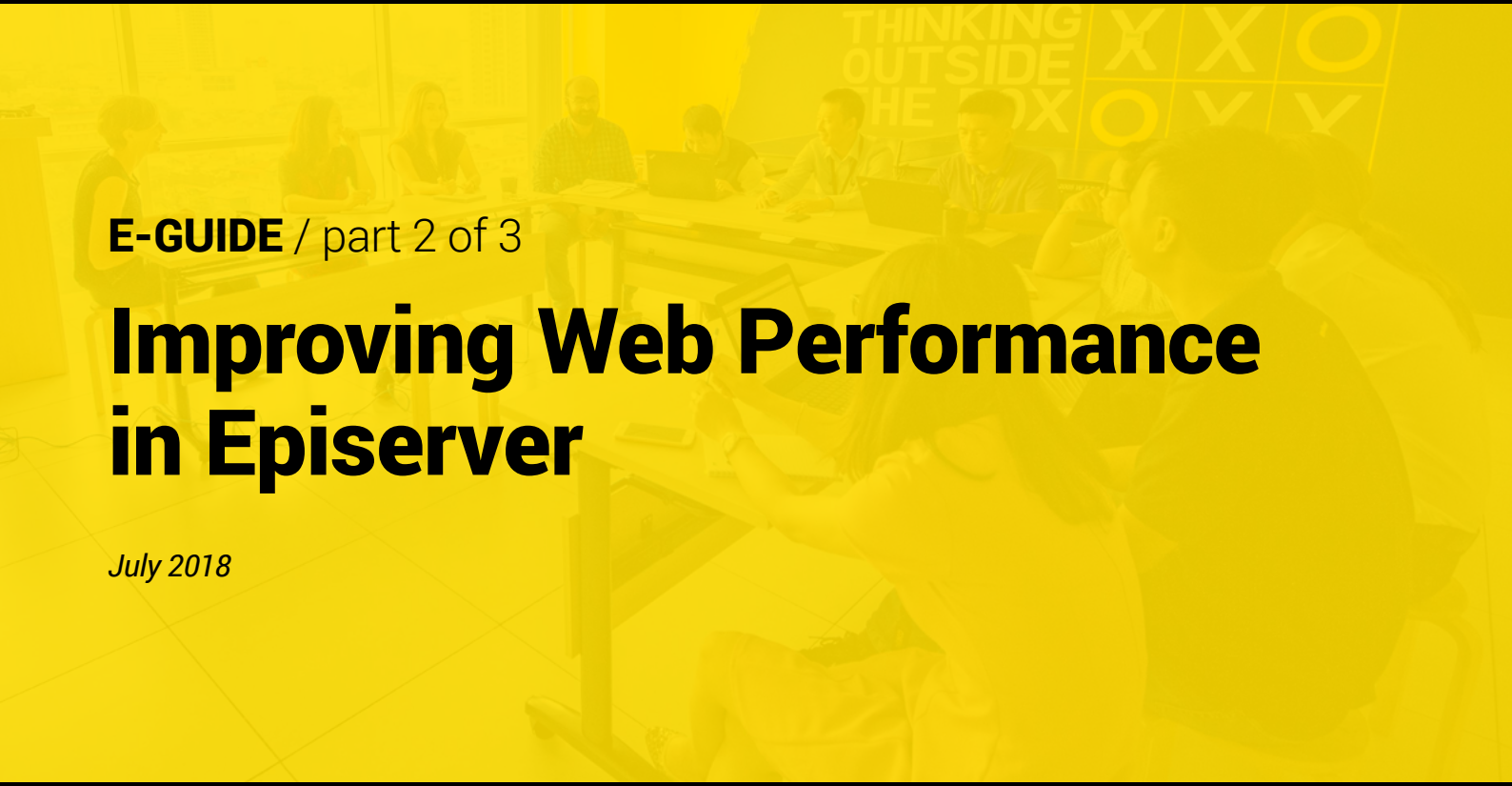
niteco
Powering Your Ambition

info@niteco.com
niteco.com



E-GUIDE / part 2 of 3

Improving Web Performance in Episerver

July 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.

Frontend Solutions

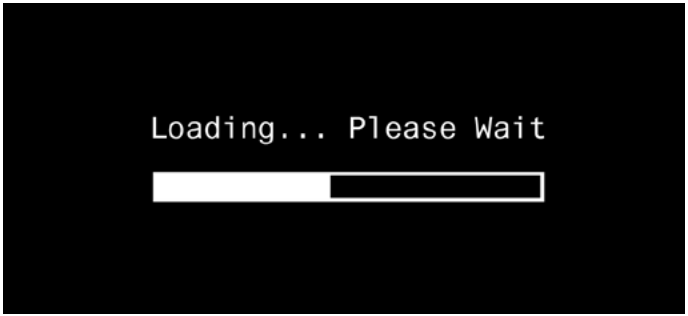
*In part 2 of our 3 part Web Performance e-guide series, we look at Frontend Solutions. * Missed part 1?* ***Download it here now.***

Load to First Interactive

Metrics like *'time to first byte'* are good to measure the loading time for a page, which is basically a combination of latency and server generation time. However, for many modern sites that feature a lot of interaction, there is usually more time being spent on loading and executing client resources. What your user wants is as short a wait as possible until they can interact with your page, and this can be done only once the content in the render critical path has been loaded and executed.

What you should focus on when reviewing performance for a solution, therefore, is another metric: Time to first interactive, describing the time that elapses until your visitor can actually do something meaningful on your site. This is the metric your page should be optimized for and it includes:

1. Loading the page.
2. Parsing the HTML.
3. Downloading required images, JavaScript, and style sheets.
4. Parsing and executing the JavaScript.
5. Parsing and executing the style sheets.

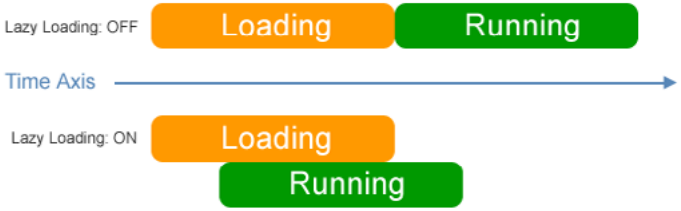


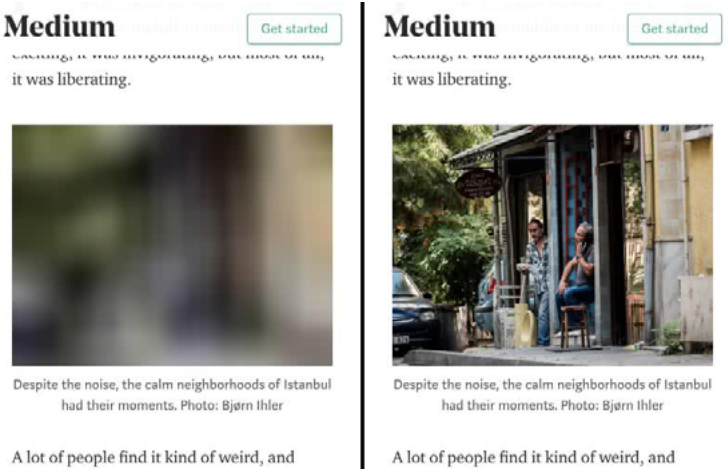
This optimization is built on one simple foundation: Making sure more important parts of the page are loaded before the more insignificant ones. In essence, ensuring that the data your site needs to become interactive is loaded first.

This also means that parts of your page that might take significant time to load could potentially be deferred to be loaded when needed. Examples of this could be to not load all images on a page before they actually appear on the user's screen, or to defer loading inventory from an external ERP system until after the main page has been loaded. It´s quite likely that the user will not look at the inventory until they have at least scanned the main part of the product page anyway.

Lazy Loading

The practice of lazy loading can help to significantly reduce the strain on the client side. It means that assets, particularly images, to be loaded for a page are prioritized according to when the user actually sees them in his browser window. Initially, only the images at the top of the page would be loaded, while the ones that remain below the visual field of the browser are kept for later. When the user scrolls down, those assets are fetched as well, meaning that the initial page load deals with much smaller file sizes. This approach is particularly useful when applied on resource-intensive pages. There are a number of JavaScript libraries that handle this, by keeping track of the viewport and copying values from an attribute like **data-src** to **src** once the image gets close to being visible. This pattern can bring down initial loading times for a page significally, especially if the page is heavy on imagery. Doing a web search on *"javascript lazy load images"* gives lots of hits to explore for anyone that is interested in delving deeper into this.





Loading and Executing Scripts

As a page is busy parsing the HTML, it will encounter scripts, either declared as inline script or as an external resource that requires the browser to download the scripts unless they are already in the cache. If not specified otherwise, these are considered to be **rendering-blocking scripts**, meaning that all other activity is put on hold while the script is fetched and executed, since it might contain important information or instructions for the parsing process. This means that just a small number of scripts which have to fetched in the middle of rendering can significantly delay the loading process. When we delve into performance tools later in this guide, we will talk more about how to detect render blocking scripts.

There are some simple solutions to lighten this load, which should be considered best practices. From the Mozilla documentation about the script element:

async: *"This is a Boolean attribute indicating that the browser should, if possible, execute the script asynchronously."*

This means that for newer browser versions that support this attribute, the page execution is not halted because of the script, meaning a smoother loading process. Dynamically inserted script using the **document.createElement** function is executed asynchronously by default, unless specifically amended with the **async** attribute set to false.

Another option is the **defer** attribute, which sets the script to be executed only after the page has finished parsing. This is especially useful for scripts that aren't actually needed for the page to work, for instance page tracking scripts.

Once again looking at the Mozilla documentation:

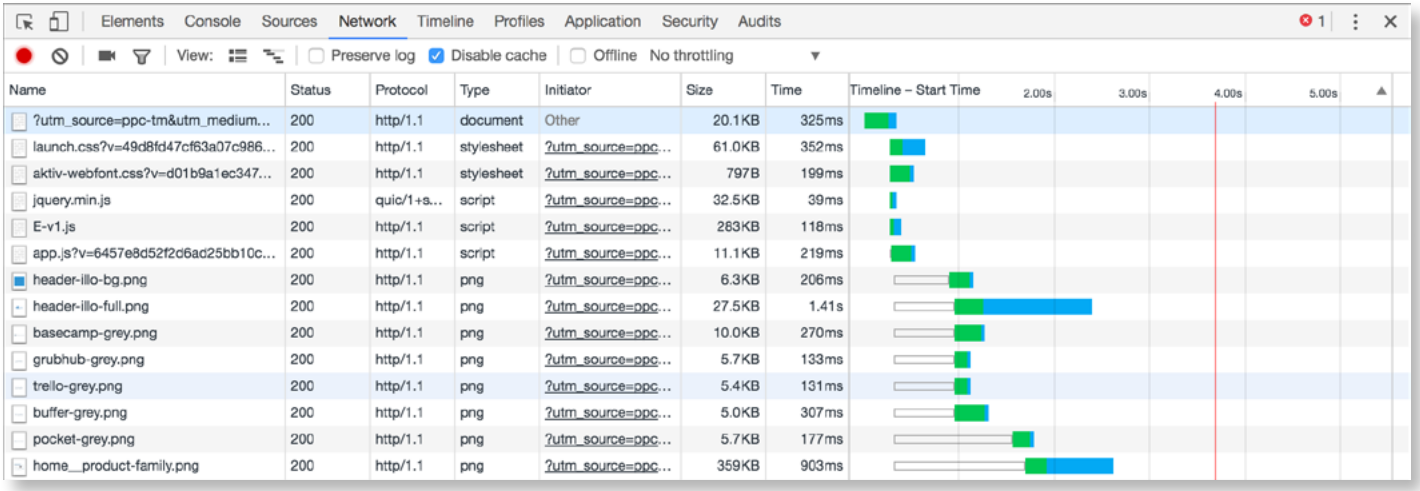
*"This Boolean attribute is set to indicate to a browser that the script is meant to be executed after the document has been parsed, but before firing **DOMContentLoaded**. Scripts with the defer attribute will prevent the **DOMContentLoaded** event from firing until the script has loaded and finished evaluating. To achieve a similar effect for dynamically inserted scripts, use async=false instead. Scripts with the defer attribute will execute in the order in which they appear in the document."*

As part of a link element, a preload value for the **rel** attribute can also substantially shorten the parsing process by starting to load scripts very early in the process. It is an instruction to load important resources right from the start of the page loading process. This ensures that they are fetched and ready to execute when the parsing procedure reaches them, meaning you don't have to wait for parsing to continue while the script is fetched. You can read more about the possibilities of the preload value here: ***https://developer.mozilla.org/en-US/docs/Web/HTML/Preloading_content***

Using HTTP/2

HTTP1, which until today has been the standard for sending web pages, was released in 1997. HTTP2, which was released in 2014, comes with a number of much-needed improvements.Support in browsers, web servers, and other parts of internet infrastructure is now established. Adapting your website to use version 2 of what is arguably the internet's most important protocol is a no-brainer. There are no real drawbacks to using it, since all major browsers support it and those that don't will revert to using HTTP/1.1 anyway. There are also a lot of performance improvements inherent in the protocol.

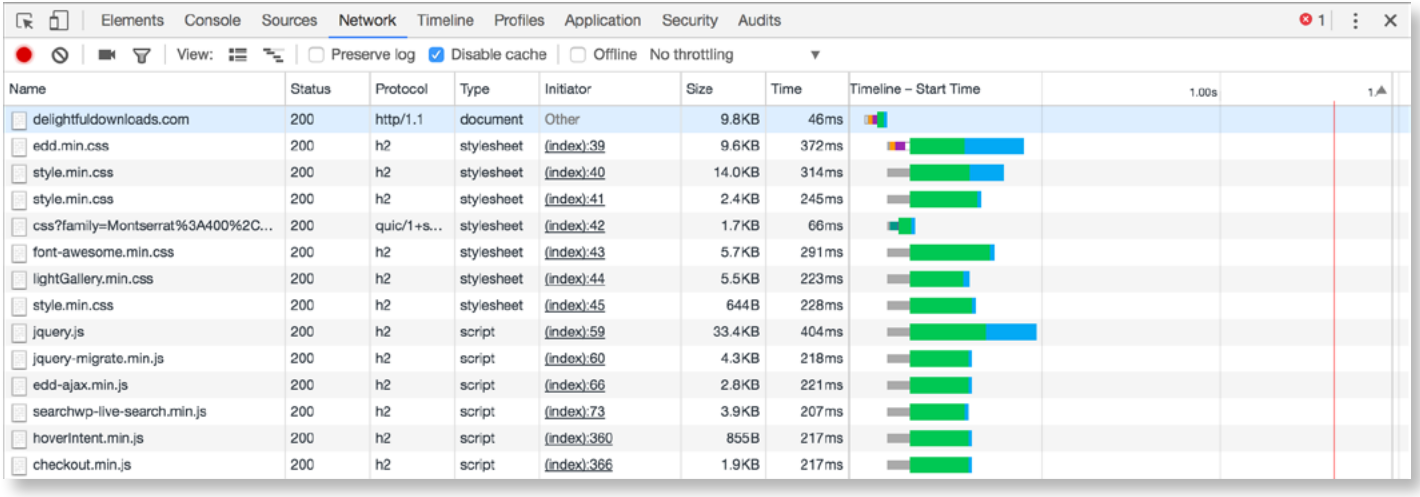
First and foremost is multiplexing, which fixes one of the biggest issues of HTTP/1.1. The old version of the protocol only allowed one request to be handled per connection, which resulted in a long queue of resources to be fetched from the server. Browsers tried to get around this problem by opening multiple connections at once, from two to eight, depending on the browser. Under this method, a standard load looked like this:



As we can see, once the page has loaded and parsing begins, a number of requests are being made almost in parallel. However, once the browser limit for requests per domain has been reached, additional assets to be downloaded have to wait their turn, with the added problem of the time it takes to open a new connection, especially when HTTPS is involved. Only when an asset is returned from the server can the browser make additional requests.

A quite usual pattern to mitigate this problem has been to separate assets on different domains; for instance, mysite.com would have the additional domains static1. mysite.com and static2.mysite.com. While this enables the browser to start downloading more assets in parallel, it comes with a cost, since there need to be more connections and additional DNS lookups have to be made.

With multiplexing, a single connection can handle multiple requests and responses at once. This means that the browser can request the download of an asset as soon as it finds it, without having to wait for an open connection. In this case, the loading process looks more like this:



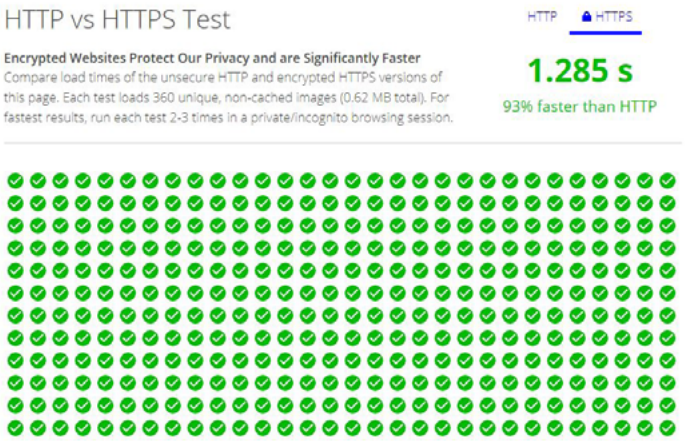
We see that all assets are downloaded immediately, as soon as they are found. We also see that the loading process is much faster as a result. The difference can be particularly large in scenarios with lots of small assets and higher latency, since the browser will have to wait for assets to be sent over the internet.

Script Bundling

A common best practice has been to keep JavaScript files separate when developing, but to bundle them into one or more larger files in production. This reduces the amount of downloaded assets that the client needs to request. This can even be applied to external scripts to even further reduce the amount of downloads to the page. This had a large impact on sites with a lot of script files running HTTP1. However, the downside to this is that a change to one small file could evict a larger bundle from the cache, since the user needs to download the entire bundled file to get the update to the changed script file.

How does this apply when using HTTP2? As multiple small files can be sent over the same connection, you might think that you should skip bundling altogether, since not all files in a bundle might be needed by a user and it would avert the problem of evicting more than the changed file when a change has been deployed. However, it turns out that a file has a certain overhead when being downloaded, meaning that lots of smaller files might not be ideal. My recommendation would be to create a few bundles, for instance one for a script framework (for instance React.js), one for your commonly used scripts, and potentially also one or more for scripts that are only used for certain pages. However, it all depends on your solution as well as the user base.

Be aware that the implementation of HTTP/2 requires you to also enable HTTPS and SSL. This is not stated in the protocol's specifications, but is required by the browsers actually supporting the newer version of the protocol. You can see a nice visual simulation of HTTP1 vs HTTP2 on the website ***https://www.httpvshttps.com/***, showing that HTTPS can actually make the loading process much faster.

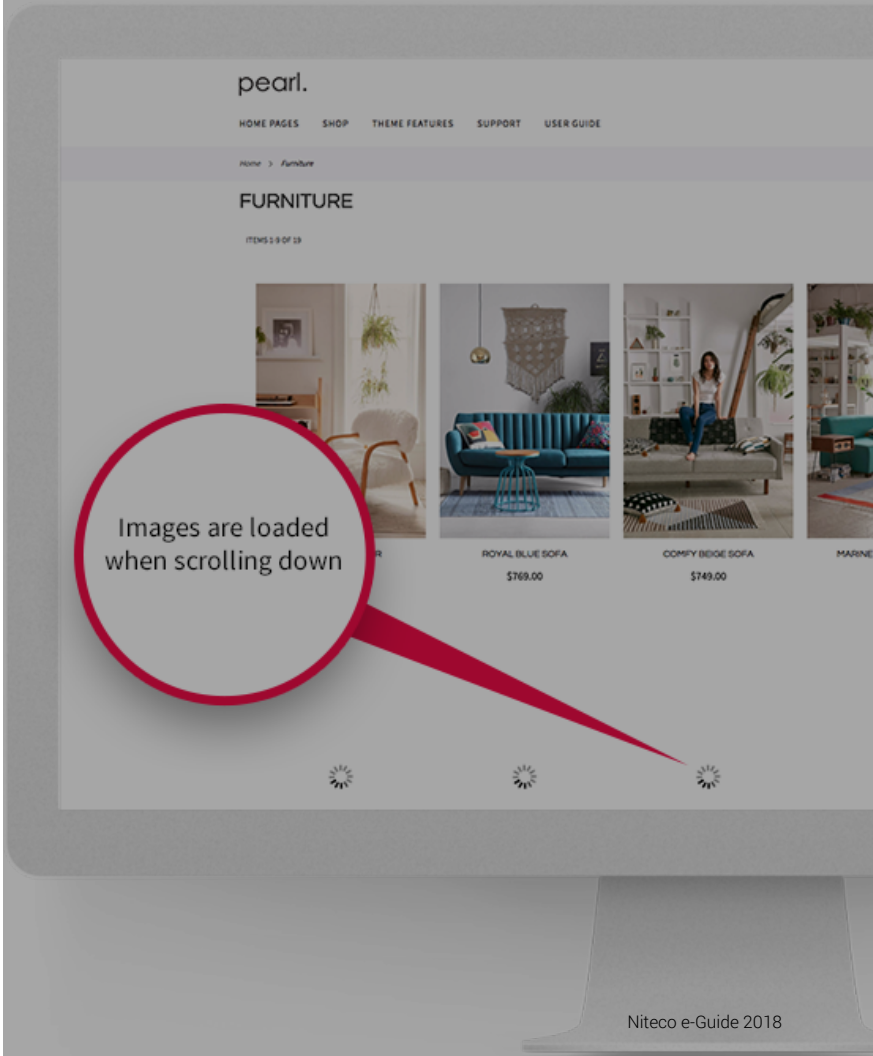


E-Commerce Specifics

With E-commerce sites being much more reliant on data pulled from the database, there are potential issues that will arise more frequently on an E-commerce site than on a traditional CMS site. With extensive product catalogues that need to be updated regularly and presented to users in varying fashion, database calls mount up substantially. In this version of the e-guide, we will not delve into Episerver Commerce that much, but I would like to highlight the serializable carts feature.

In a recent update, Episerver also introduced serializable carts, which store a user's shopping cart as JSON, potentially improving performance significantly.

- ***https://world.episerver.com/blogs/Son-Do/Dates/2017/1/introduce-serializablecart-mode/***
- ***https://world.episerver.com/blogs/andreas-j/dates/2017/6/benchmarking-episerver-serializable-carts2/***



Performance Testing

How to measure performance

In order to improve your site's performance, you need to find out how your site currently performs. Once this has been done, you can start prioritizing and improving the site. To check the performance of a site, there are a number of tools that focus on testing from an end user's perspective. We will delve into server performance tools later.

We can group these into two kinds of monitoring tools that can help give you an overview of how a user perceives your site*. Passive tools are used to manually test performance, active tools automatically monitor performance.Some of the most well-known passive tools are Web Page Test and Google Lighthouse, both of which give you a clean overview of their test results.

**Actually, the tools will give you a lot of measurements which will give you a good idea of how the user will perceive the site, but this can never replace real end-user testing.*

Web Page Test

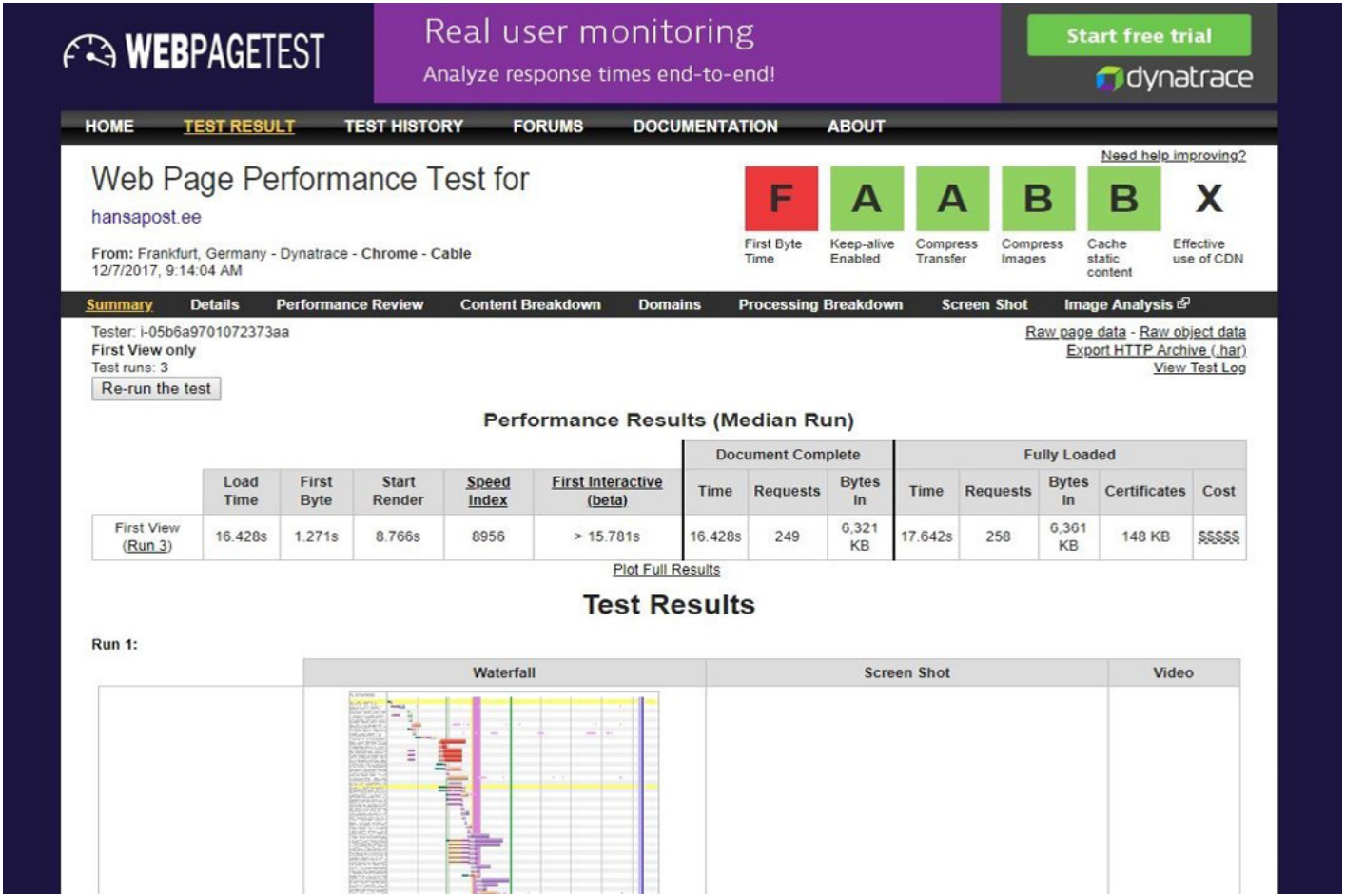
Webpagetest (***https://www.webpagetest.org/***) is a free online tool that makes it possible to run performance test on any publically available site. The tool, which is sponsored by industry companies, like CMS and CDN vendors, makes it possible to test the performance of a site from any of a large range of nodes around the world. You can select the browser you want to simulate as well as set the download speed. This is great, for instance when you want to test how a global site performs in different areas of the world. You can save the link to a test that you run and share it with others, and the tool churns out lots of good information, for instance performance breakdowns and recommendations, as well as videos showing how the site load works. The tool runs tests against the URL three times by default, to be able to check for variations in performance, for instance if the first request requires additional things, like loading content from the database or triggering image resizing on demand on the first request.

Pros:

- Free and easy to use.
- Relatively easy to learn.

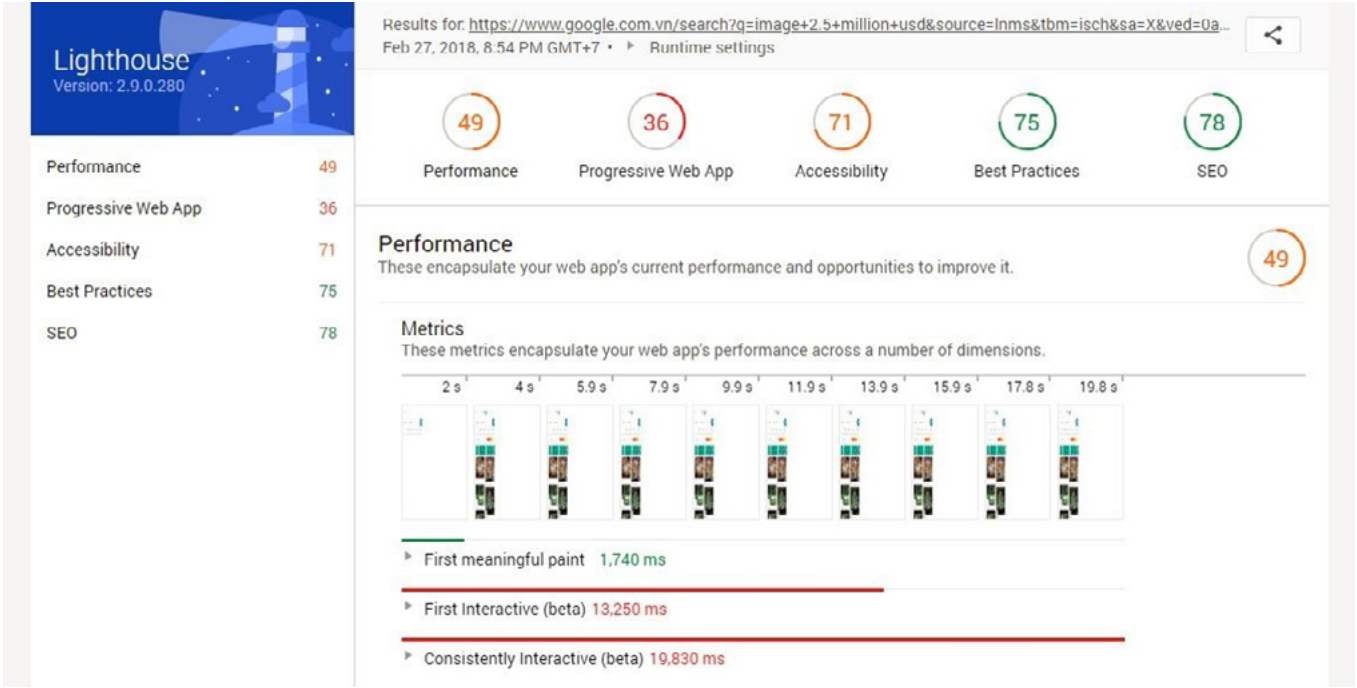
Cons:

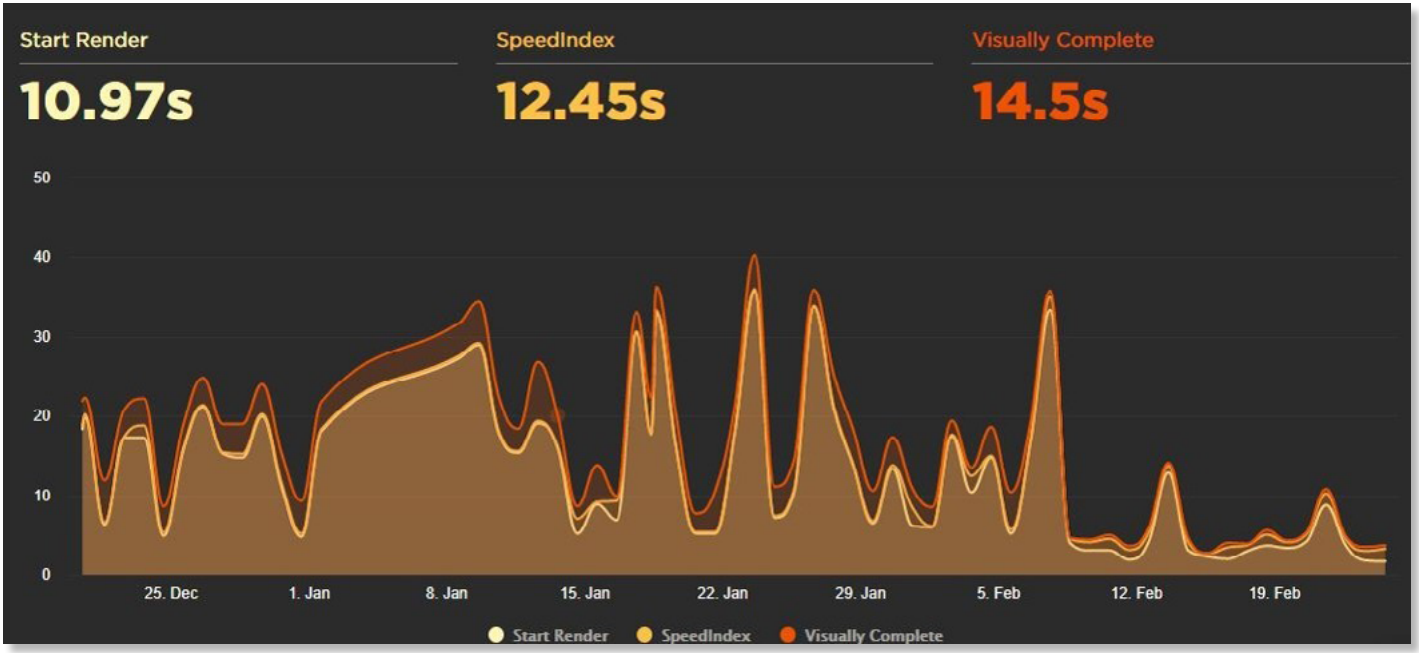
- Since the tool is free, there can sometimes be a queue of requests to their servers, meaning that it might take a little time until you can see your results.
- You need to actively run the tool and save any performance measurements that you want to use to compare performance over time.
- You can only run it on publically available domains, which might prevent you from accessing an internal site or a test or staging environment.


WEBPAGETEST
Real user monitoring
Analyze response times end-to-end!
Start free trial
dynatrace
HOME
TEST RESULT
TEST HISTORY
FORUMS
DOCUMENTATION
ABOUT
Need help improving?
Web Page Performance Test for
hansapost.ee
From: Frankfurt, Germany - Dynatrace - Chrome - Cable
12/7/2017, 9:14:04 AM
F
First Byte Time
A
Keep-alive Enabled
A
Compress Transfer
B
Compress Images
B
Cache static content
X
Effective use of CDN
Summary
Details
Performance Review
Content Breakdown
Domains
Processing Breakdown
Screen Shot
Image Analysis
Tester: i-05b6a9701072373aa
First View only
Test runs: 3
Re-run the test
Raw page data - Raw object data
Export HTTP Archive (.har)
View Test Log
Performance Results (Median Run)
Document Complete
Fully Loaded
Load Time
First Byte
Start Render
Speed Index
First Interactive (beta)
Time
Requests
Bytes In
Time
Requests
Bytes In
Certificates
Cost
First View (Run 3)
16.428s
1.271s
8.766s
8956
> 15.781s
16.428s
249
6,321 KB
17.642s
258
6,361 KB
148 KB
$$$$$
Plot Full Results
Test Results
Run 1:
Waterfall
Screen Shot
Video

Google Lighthouse

Google Lighthouse is a free extension to Google Chrome that makes it possible to track a request being made through the browser. Though this means that it is connected to how the Google Chrome browser behaves, it gives a lot of valuable insight. I usually run this tool when improving a site, since it makes it possible to undergo the test-and–improvecycle in an easy way in a local development environment. ***https://developers.google.com/web/tools/lighthouse/***





Speedcurve

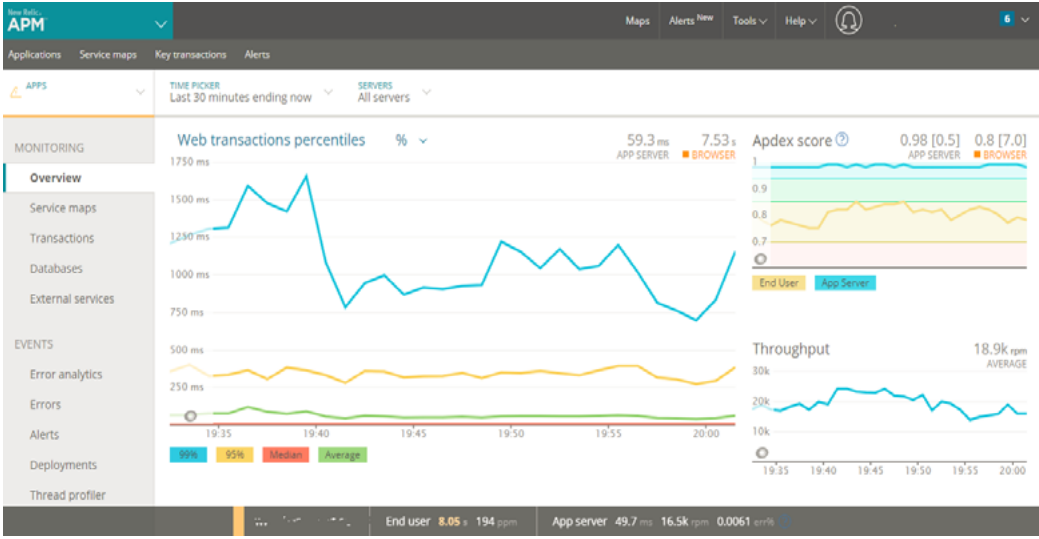
The active tool Speedcurve offers automatic tracking using a private cloud network of Web Speed Test servers. Though the tool includes a lot of nice features, the most obvious and easy to use is to automatically track performance over time. This is great, since it enables you to always keep an eye on the most important tracking measurements. Perhaps a new release made the site faster, or worse, slower. You can even configure performance budgets and fire off alarms when the performance is not up to the standards you have defined. There are several performance measurements to choose from. With its API, it also enables custom tracking. ***https://www.speedcurve.com***

New Relic

New Relic is a professional tool that is currently included in Episerver DXC (though it is being replaced with Application Insights sincethe Spring 2018 Episerver Ascend events). It includes lots of functionality, for instance:

- Active end user performance tracking (similar to Speedcurve).
- Infrastructure health monitor checking, which enables you to see CPU and memory for the application and database servers.
- The ability to see the controllers taking up the most time and to drill down into these to see which code takes the longest time to execute.
- List of most occurring errors.
- Email notification when the application does not perform well.

The New Relic license model is separated into several different parts, so you can choose to only use the Application Monitoring tool.



Screenshot showing the response time overview view in New Relic.

In part 3 of our e-Guide, we will tell you where to get started when you've decided to work on your website's performance. In addition, we show you a real-life example of what good performance tuning can do.

POWERING YOUR EPISERVER AMBITION

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

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

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

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

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 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

niteco.com

Niteco E-guide / NITECO1807