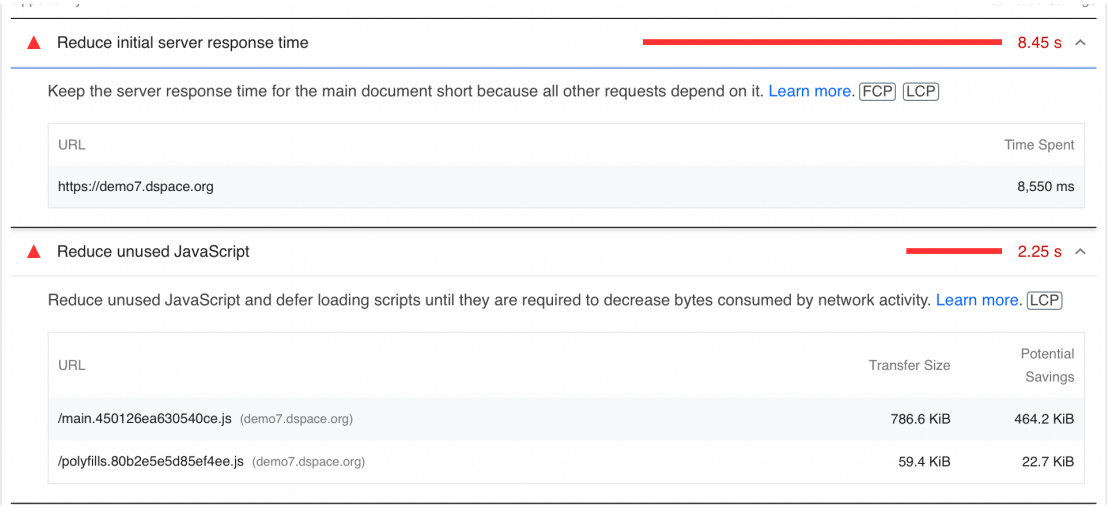
DSpace 7 UI Optimization Analysis

During the DevMtg on Oct 13, 2022, we began a discussion on improving the performance of the DSpace 7 UI, especially in terms of **initial load.** This page is meant to gather information / notes for further analysis

Overview

Goal is to improve the initial load of the UI per user reports and Google Lighthouse analysis (see screenshot)

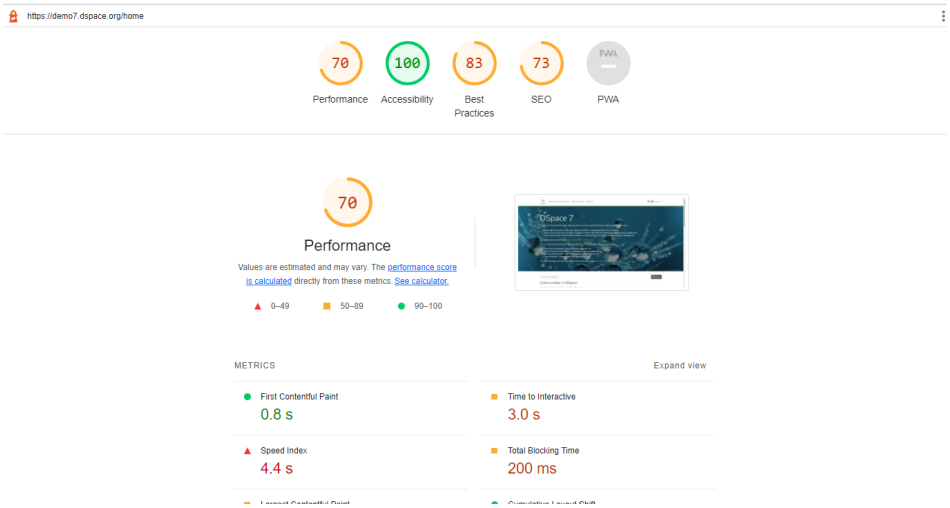


DSpace 7 UI Infrastructure Examples

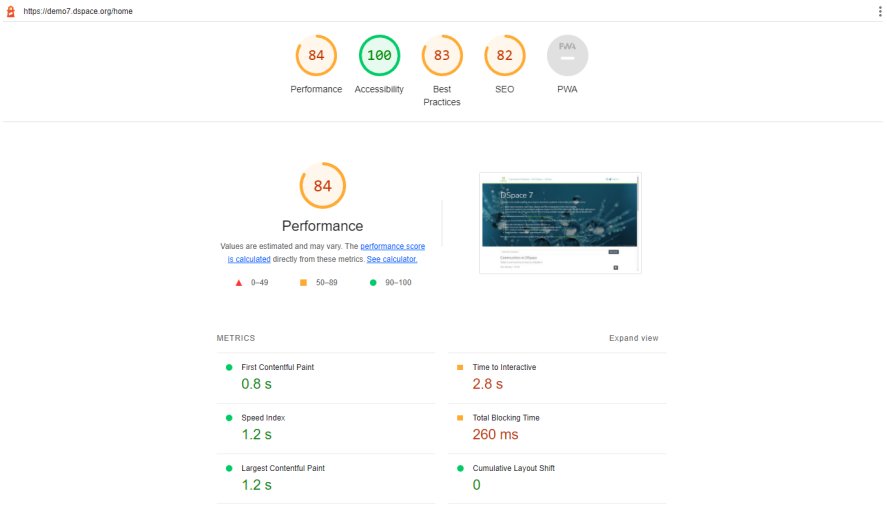
DSpace 7 Demo Site UI infrastructure

TODO: Art Lowel (Atmire) or Atmire provide details on the infrastructure behind https://demo7.dspace.org/ (DSpace 7 Demo UI), especially in terms of the Node.js setup/configuration and setup/configuration of proxies.

NOTE: Here's the Google Lighthouse performance as of Oct 13, 2022 (Running DSpace 7.4):



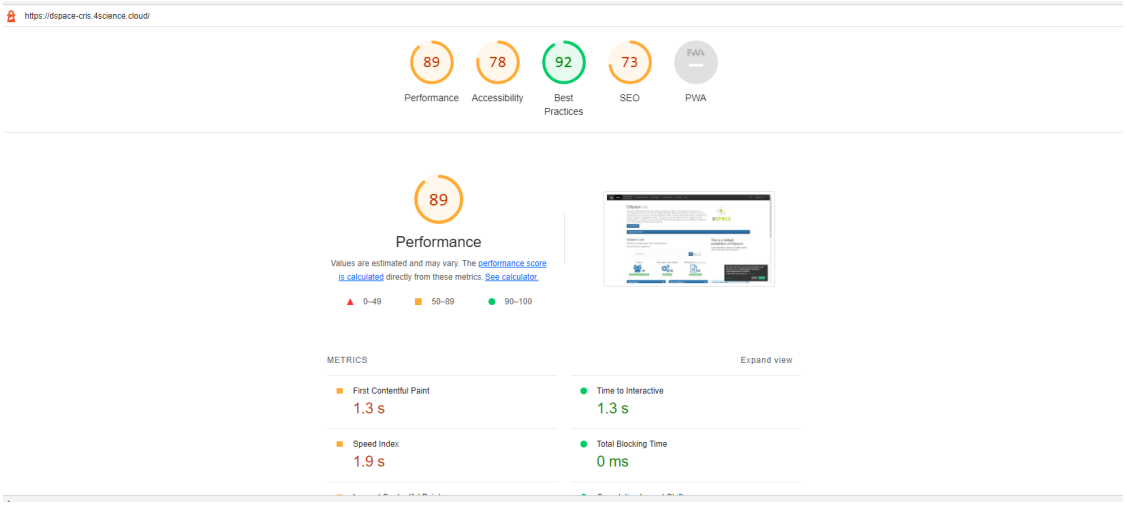
UPDATE: Here's the Google Lighthouse performance as of Feb 15, 2023 (Running DSpace 7.5, with "page caching" enabled for anonymous users):



DSpace-CRIS 7 Demo Site UI infrastructure

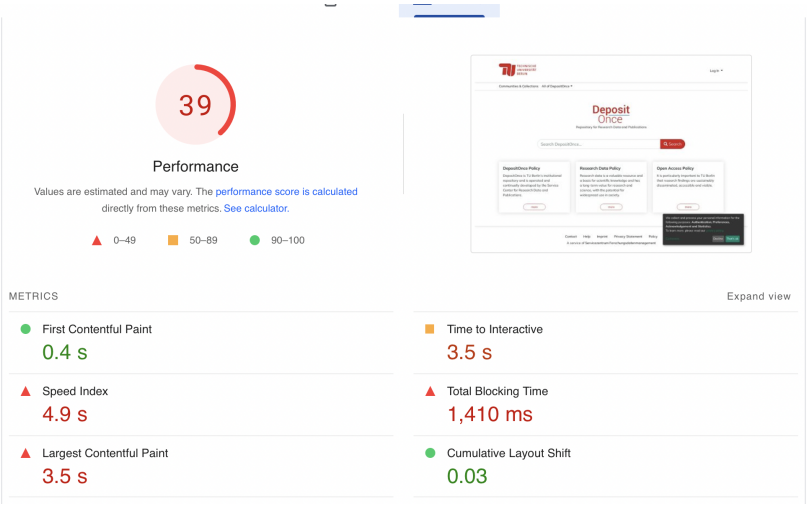
TODO: Andrea Bollini (4Science) or 4Science provide details on the infrastructure behind https://dspace-cris.4science.cloud/ (DSpace-CRIS 7 Demo UI), especially in terms of the Node.js setup/configuration and setup/configuration of proxies.

NOTE: On Oct 13, 2022, DSpace-CRIS 7 demo site has better Performance results from Google Lighthouse than DSpace 7 Demo site:



TU Berlin DSpace 7 UI infrastructure

Google Lighthouse performance results for https://depositonce.tu-berlin.de on October 14, 10:20am:



Our servers run at the university's IT department in an Open Stack environment with these specs:

Server	Cores	RAM
Backend	16	64GB
Frontend	8	32GB
PostgreSQL	4	16GB
SOLR	2	8GB

This is our PM2 configuration file:

dspace-ui.json

```
{
  "apps": [
    {
      "name": "dspace-ui",
      "cwd": "/srv/dspace-ui-deploy",
      "script": "dist/server/main.js",
      "node_args": "--max_old_space_size=8192",
      "env": {
        "NODE_ENV": "production",
        "DSPACE_REST_SSL": "true",
        "DSPACE_REST_HOST": "api-depositonce.tu-berlin.de",
        "DSPACE_REST_PORT": "443",
        "DSPACE_REST_NAMESPACE": "/server"
      }
    }
  ]
}
```

The frontend runs behind an Apache proxy, as described in the Installing DSpace page.

The backend also runs behind an Apache proxy, as described in the Installing DSpace page. I increased the Java memory significantly:

```
Environment="JAVA_OPTS=-Djava.awt.headless=true -Xmx8G -Xms2G -Dfile.encoding=UTF-8"
```

Update:

We had some success with running more than one instance of the frontend using PM2 Cluster Mode. It made our response times significantly better. But we are still far from an acceptable performance. This is our new PM2 configuration file:

dspace-ui.json

```
{
  "apps": [
    {
      "name": "dspace-ui",
      "cwd": "/srv/dspace-ui-deploy",
      "script": "dist/server/main.js",
      "instances" : 4,
      "exec_mode" : "cluster",
      "node_args": "--max_old_space_size=4096",
      "env": {
        "NODE_ENV": "production",
        "DSPACE_REST_SSL": "true",
        "DSPACE_REST_HOST": "api-depositonce.tu-berlin.de",
        "DSPACE_REST_PORT": "443",
        "DSPACE_REST_NAMESPACE": "/server"
      }
    }
  ]
}
```

Optimization Strategies

Node.js configuration for multi-threading

Andrea Bollini (4Science) mentioned in the meeting on Oct 13, 2022 that 4Science has discovered that the default setup of Node.js can be limiting for sites with a lot of users simply because Node.js is single threaded. He noted that 4Science has found ways to configure Node.js to better support many users by allowing Node.js to use all your CPU, etc.

4Science notes: Optimization of the hosting environment is one of the area where the competition across service providers is higher. At 4Science we have and we are investing a lot in tools to monitor, enhance and benchmark different setup and these findings constitute one of our competitive advantages. In the spirit of the open source community and in the respect of the investment done, we are happy to share some of these findings with the community.

Node.js follows a single-thread model. This mean that when a request trigger the server side rendering (SSR) other concurrent requests will need to wait until the previous requests has been served. Depending on the capacity of the server and the speed of all the other involved components (REST, DB, SOLR) this could vary from 1 seconds or less to more than 2-3 seconds (or much more). PM2 has a cluster mode for Node.js that start multiple nodejs instances distributing requests among them allowing to use more than one physical or virtual CPUs, see https://pm2.keymetrics.io/docs/usage/cluster-mode/

without this setting any additional CPU available on the frontend/angular server will be just ignored and only 1 CPU will be really used most of time.

In general we found important to provide more CPU to the angular/node components and to the SOLR component than to the REST API server. Memory is more relevant for the REST API/tomcat and for SOLR. For SOLR it is important to avoid swapping, see https://solr.apache.org/guide/8_11/taking-solr-to-production.html#ulimit-settings-nix-operating-systems

Our usual setup adopts multiples VMs, one VM for each DSpace component (reverse proxy, REST, Angular, PostgreSQL, solr, ...) this help to track down bottlenecks and to measure resource loads. One important aspect is also the reverse proxy, we found nGINX usually to be more performant than apache when no special configuration are applied to both. Keep in mind that the reverse proxy will be the final bottleneck for everything as all the requests (REST /SSR) will pass in it. If the CPUs of the server are doing other stuff the reverse proxy will be not able to fulfill the requests and the request will be stuck in the queue. We have observed sometimes very slow response time from the REST due to the fact that all the CPUs where already used by the reverse proxy / SSR part that were not able to free up such resources because they were waiting for requests sent to the REST API that never reach tomcat (as they were queued in the reverse proxy...)

In our opinion, If you really want to go with a single all in one server you should use not less than 8vCPU and 32GB of RAM

ACTION: Updated all DSpace Documentation to include this hint, especially Performance Tuning DSpace.

Minimize size of main.js

Team needs to investigate if there are ways to further minimize the size of the main.js file (noted by Lighthouse as being too large) via lazy loading or similar strategies.

In 7.5, work on this was completed in https://github.com/DSpace/dspace-angular/issues/1921 (see PRs attached/linked to this ticket)

Examples: https://christianlydemann.com/the-complete-guide-to-angular-load-time-optimization/

This **might** require optimizing pages which users are most likely to first access the application. Namely:

- Homepage
- Item splash pages
- (Possibly others? Community/Collection splash pages?)

Page Caching

Improvements to page caching was completed in 7.5, see https://github.com/DSpace/dspace-angular/pull/2033

Tools

webpack-bundle-analyzer

Can be used to visualize what is taking up a lot of space in main.js and other bundles

https://www.npmjs.com/package/webpack-bundle-analyzer

Running for DSpace:

- Install it (globally): `npm install -g webpack-bundle-analyzer`
- Run a build and generate stats.json: `yarn build:stats`
- Start it up! `webpack-bundle-analyzer .\dist\browser\stats.json`

BundlePhobia.com

Useful to find the normal size of dependencies & links you to alternative libraries which may be smaller: https://bundlephobia.com/

It also details which libraries are "tree shakable" and which are not.