Open in app ↗

Medium Search Write 1

BlackRock Engineering

Improving monorepo build times using Nx

Gary describes how his team is leveraging Nx features to reduce pipeline execution times for their monorepo. He deep dives into Nx distributed caching and the process through which he was able to make significant improvements to the team's build and test times.

BlackRockEngineering Follow 8 min read · Sep 1, 2023

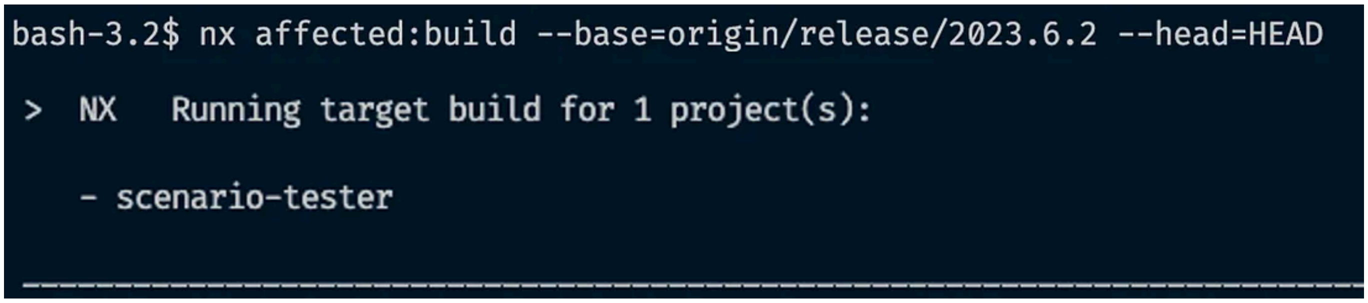
59

> **By: Gary Dexter**, Lead Engineer in Aladdin Wealth Tech, BlackRock

In a recent blog post, "Don't get tangled in your code," we spoke about how we use Nx to manage a monorepo containing a large number of applications and shared functionality across libraries, as well as some of the challenges that arise from this approach. Recently we started using more Nx features to help reduce our overall pipeline execution times.

Nx affected

As we discussed in our previous blog post, we use Nx's dependency graph and `nx affected` to execute tasks only on the applications and libraries that are affected by the changes in the current pull request. Bypassing the PR's target branch as the `base` argument to `nx affected` commands, Nx will figure out what has changed. Then, based on the dependency graph, it automatically runs the tasks for only the affected projects.



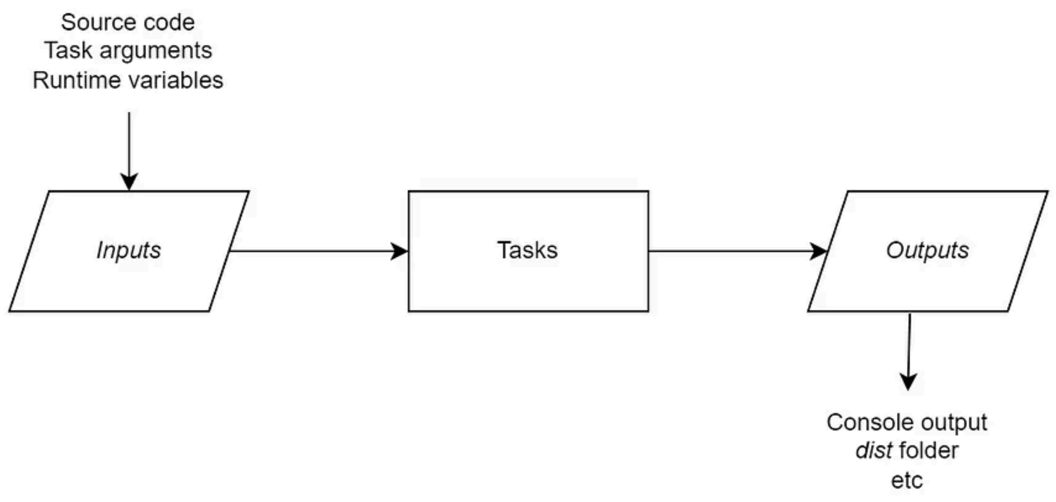
```
bash-3.2$ nx affected:build --base=origin/release/2023.6.2 --head=HEAD

 >  NX   Running target build for 1 project(s):

    - scenario-tester

 ------------------------------------------------------------------------
```

This method reduced build and test execution times in some cases, but as our code is tightly intertwined with several shared libraries, and many pull requests change those libraries, we often found that most of our projects were still "affected" — and therefore were still built or tested. This also only worked for our quality builds; for publish builds, we still needed to build and test everything from scratch. We needed something better.

Task caching

Nx uses a computational caching system. If you are running a task that was previously executed, Nx restores the results of running that task from the cache instead of executing it again.

In other words, if you run a task with the same **inputs** as one you've run before, Nx will restore that task's **outputs** from the cache:

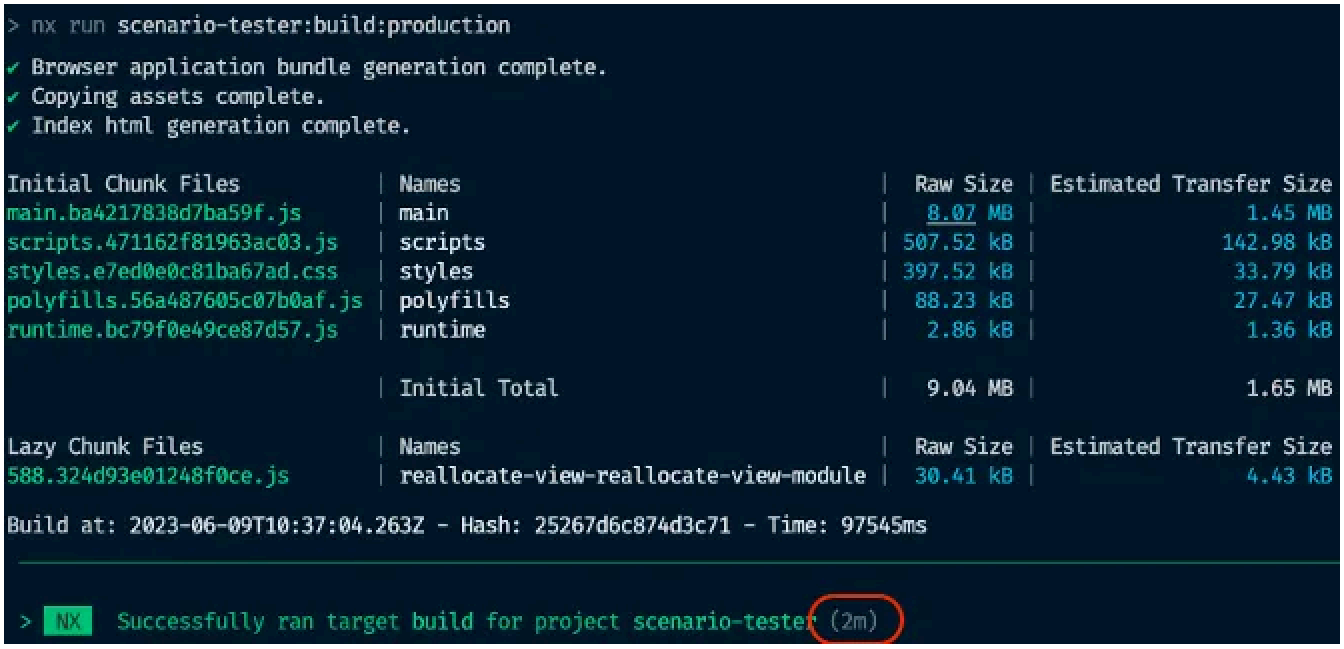


For more details on how the Nx cache works, see their documentation here.

Different tasks will require different inputs. For example, you may want a fresh run of project builds, tests and linting if any of the source code changes, but just a fresh run of tests if you change your Jest config, or just linting if your ESLint config changes. This can all be configured in your `nx.json` file, but in our case, we were happy with the defaults.

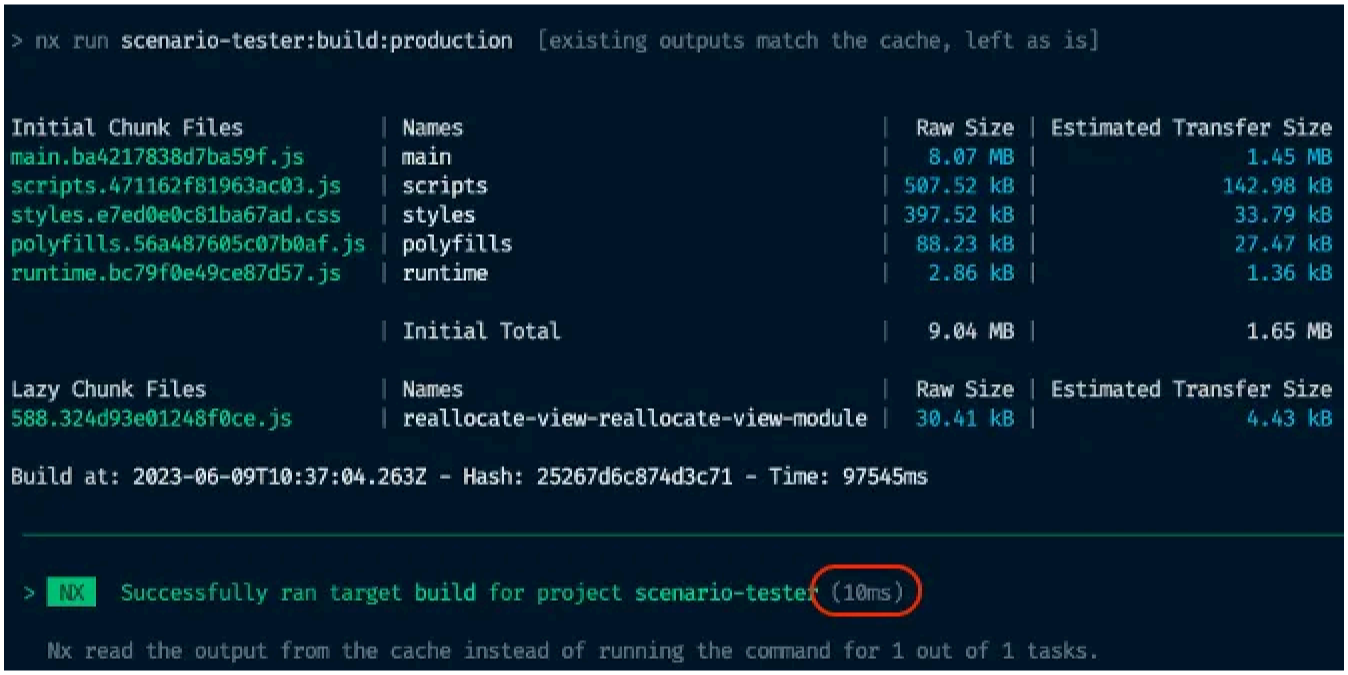
Similarly, you may want to store different outputs in the cache for different tasks. By default, Nx will store the `dist` folder for application builds, but for tests, we had to make sure that coverage files were stored too — this is so the Quality Gate will still pass if tests are not executed for every project. Each project has its own configuration file in which we can specify additional output — here we can add things like coverage files for the scenario-tester app so they will also be restored from the cache.

Let's see this in action. If we build the scenario-tester project from scratch, we see it takes a couple of minutes:



```
> nx run scenario-tester:build:production
✓ Browser application bundle generation complete.
✓ Copying assets complete.
✓ Index html generation complete.

Initial Chunk Files           | Names                                  |  Raw Size | Estimated Transfer Size
main.ba4217838d7ba59f.js      | main                                   |   8.07 MB |                 1.45 MB
scripts.471162f81963ac03.js   | scripts                                | 507.52 kB |               142.98 kB
styles.e7ed0e0c81ba67ad.css   | styles                                 | 397.52 kB |                33.79 kB
polyfills.56a487605c07b0af.js | polyfills                              |  88.23 kB |                27.47 kB
runtime.bc79f0e49ce87d57.js   | runtime                                |   2.86 kB |                 1.36 kB

                              | Initial Total                          |   9.04 MB |                 1.65 MB

Lazy Chunk Files              | Names                                  |  Raw Size | Estimated Transfer Size
588.324d93e01248f0ce.js       | reallocate-view-reallocate-view-module |  30.41 kB |                 4.43 kB

Build at: 2023-06-09T10:37:04.263Z - Hash: 25267d6c874d3c71 - Time: 97545ms

 > NX   Successfully ran target build for project scenario-tester (2m)
```

Now let's delete the `dist` folder and run the build again. This time the task is retrieved from the cache instead. All the outputs — the `dist` folder and the console output — are restored from the cache, and this happens in 10 *milliseconds* instead of two minutes.

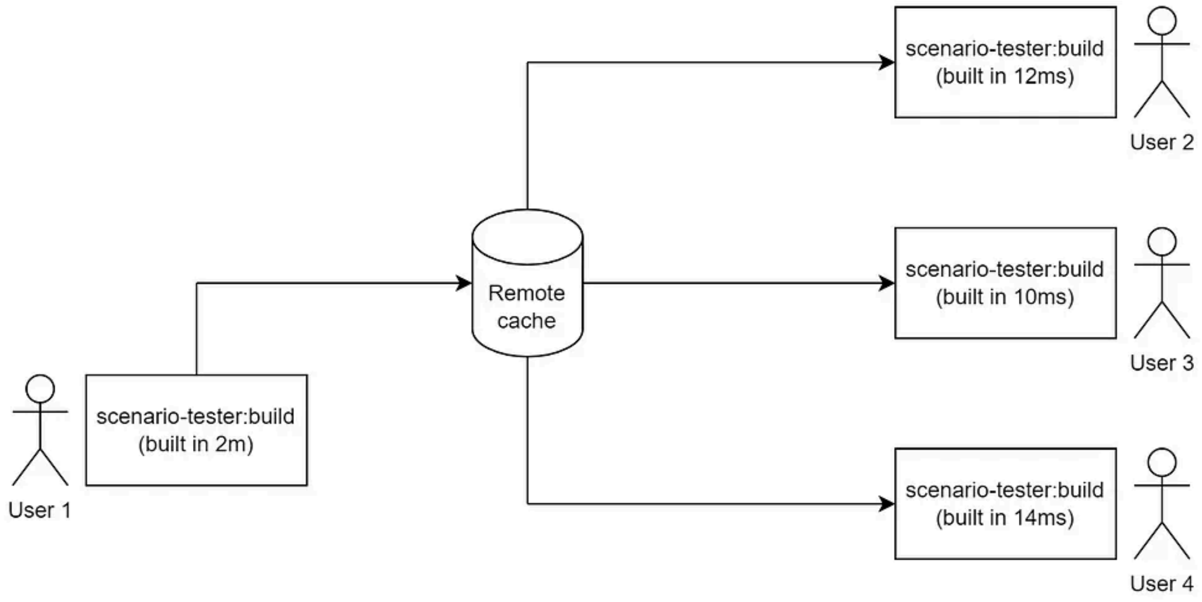


```
> nx run scenario-tester:build:production  [existing outputs match the cache, left as is]

Initial Chunk Files             | Names                                  |  Raw Size | Estimated Transfer Size
main.ba4217838d7ba59f.js        | main                                   |   8.07 MB |                 1.45 MB
scripts.471162f81963ac03.js     | scripts                                | 507.52 kB |               142.98 kB
styles.e7ed0e0c81ba67ad.css     | styles                                 | 397.52 kB |                33.79 kB
polyfills.56a487605c07b0af.js   | polyfills                              |  88.23 kB |                27.47 kB
runtime.bc79f0e49ce87d57.js     | runtime                                |   2.86 kB |                 1.36 kB

                                | Initial Total                          |   9.04 MB |                 1.65 MB

Lazy Chunk Files                | Names                                  |  Raw Size | Estimated Transfer Size
588.324d93e01248f0ce.js         | reallocate-view-reallocate-view-module |  30.41 kB |                 4.43 kB

Build at: 2023-06-09T10:37:04.263Z - Hash: 25267d6c874d3c71 - Time: 97545ms

 ————————————————————————————————————————————————————————————————————————————————————————————

 >  NX   Successfully ran target build for project scenario-tester (10ms)

    Nx read the output from the cache instead of running the command for 1 out of 1 tasks.
```

When running builds and tests across every one of our 100+ projects, this has a huge impact on the amount of computation time. Great! But so far this is only working locally. We need a way to get these benefits in our CI pipelines.

Sharing the cache

Nx is designed to allow the cache to be shared or “distributed” across multiple machines. This means that rather than storing the cache on your local machine, it is stored somewhere central that can be accessed by multiple clients. So, for example, once one person has built an app, other people can run the same build and it will be retrieved from the cache.



Nx has its own distributed cache implementation — both a freemium cloud-based version and a self-hosted on-prem version. While either of these solutions would work, due to privacy and cost constraints, neither are currently an option for our team.

Luckily, Nx also provides a public API to build custom cache implementations, and there are several npm libraries that simplify this process. This makes it possible to share the cache to various places, such as Azure, AWS, or even just a file system.

One library, nx-remotecache-custom, creates an Nx task runner using three user-defined functions to interact with the cache:

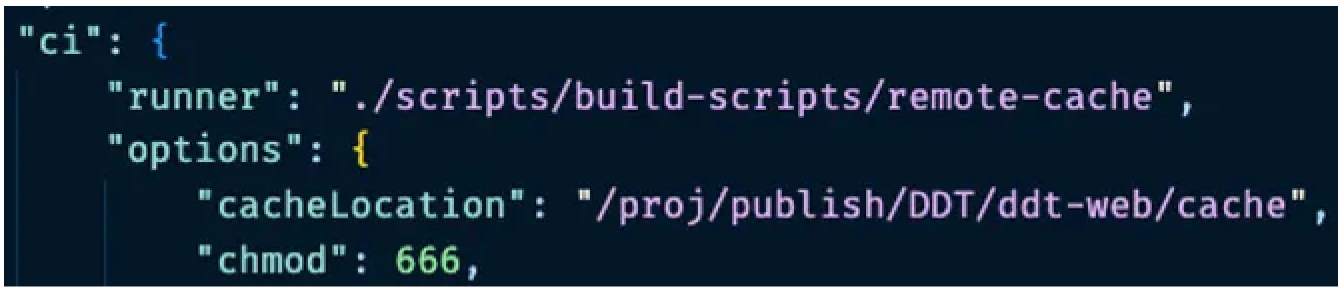
- one to store a file
- one to check if a file exists
- one to retrieve a file

We can use this to create a custom distributed cache that our CI pipelines will have access to.

How we implemented the cache

We can take what we've learned so far about the Nx cache and combine it with what we know about our Azure DevOps pipelines and create a cache that can be shared across our CI builds.

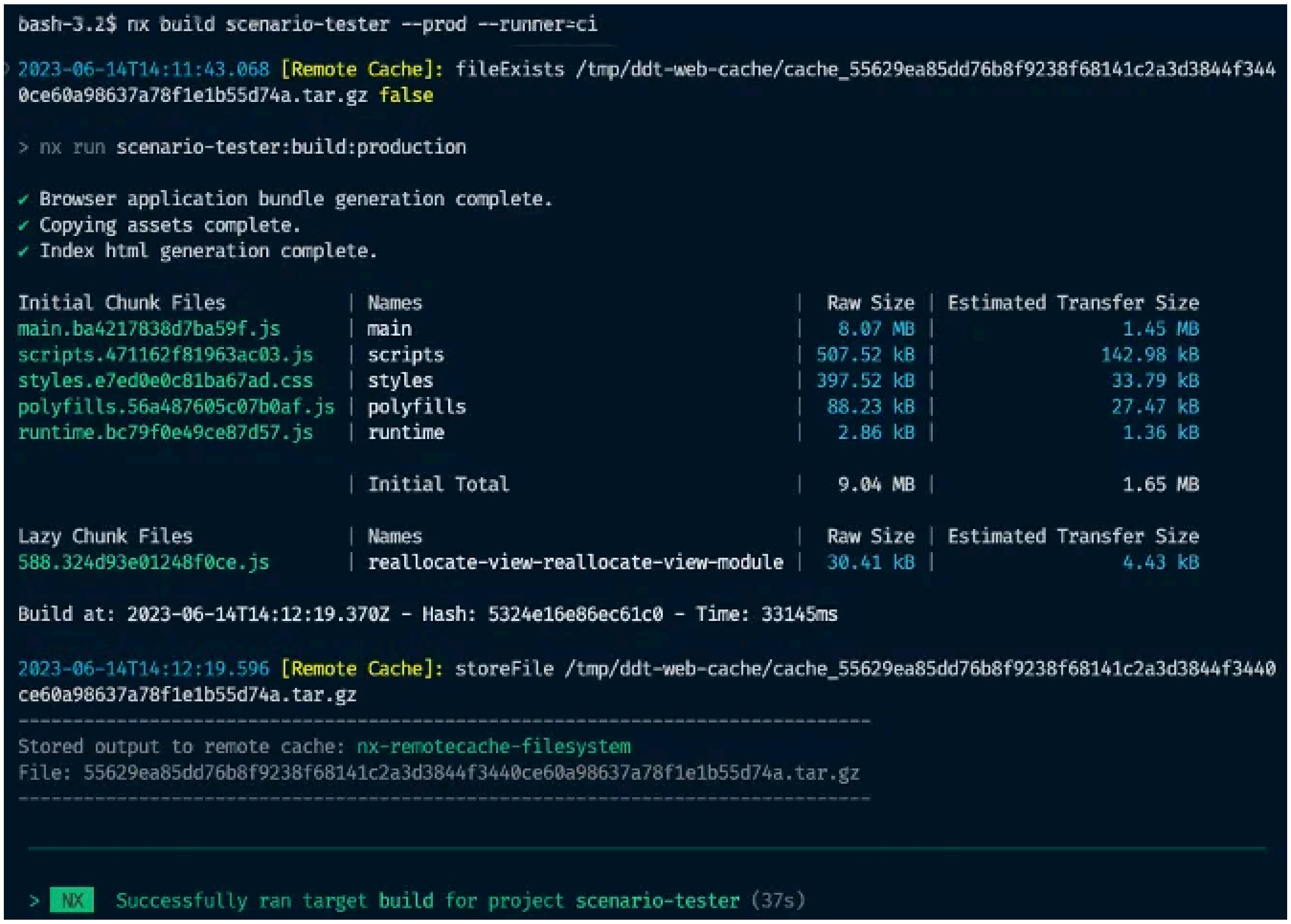
For our POC, we took advantage of the fact that all agents in our builds have access to a common, shared directory. We could then write a custom script to define the caching functions using `nx.json` to configure the task runner and specify options like the cache location and file permissions.



```
"ci": {
    "runner": "./scripts/build-scripts/remote-cache",
    "options": {
        "cacheLocation": "/proj/publish/DDT/ddt-web/cache",
        "chmod": 666,
```

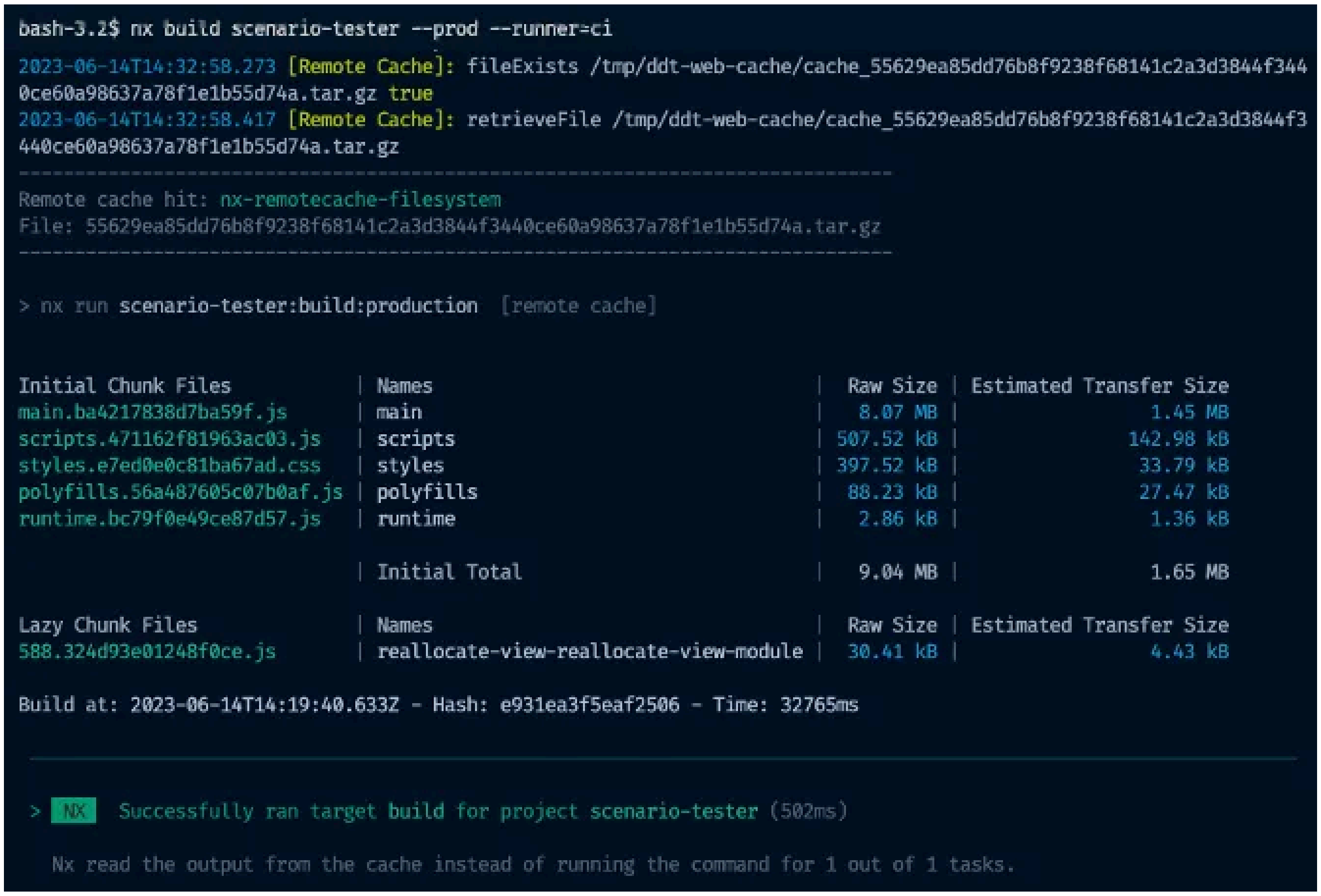
The cache functions simply use the Node filesystem to perform the actions by using `createWriteStream` to write the outputs to the cache location (compressed into a single tarball), `pathExists` to check if the cache file exists, and `createReadStream` to retrieve the file after a cache hit.

Now all we need to do is tell our CI builds to use the new "ci" task runner, and it will check the remote cache location, build the app, then store the results:



```
bash-3.2$ nx build scenario-tester --prod --runner=ci

2023-06-14T14:11:43.068 [Remote Cache]: fileExists /tmp/ddt-web-cache/cache_55629ea85dd76b8f9238f68141c2a3d3844f344
0ce60a98637a78f1e1b55d74a.tar.gz false

> nx run scenario-tester:build:production

✔ Browser application bundle generation complete.
✔ Copying assets complete.
✔ Index html generation complete.

Initial Chunk Files           | Names                                |  Raw Size | Estimated Transfer Size
main.ba4217838d7ba59f.js      | main                                 |   8.07 MB |                 1.45 MB
scripts.471162f81963ac03.js   | scripts                              | 507.52 kB |               142.98 kB
styles.e7ed0e0c81ba67ad.css   | styles                               | 397.52 kB |                33.79 kB
polyfills.56a487605c07b0af.js | polyfills                            |  88.23 kB |                27.47 kB
runtime.bc79f0e49ce87d57.js   | runtime                              |   2.86 kB |                 1.36 kB

                              | Initial Total                        |   9.04 MB |                 1.65 MB

Lazy Chunk Files              | Names                                |  Raw Size | Estimated Transfer Size
588.324d93e01248f0ce.js       | reallocate-view-reallocate-view-module |  30.41 kB |                 4.43 kB

Build at: 2023-06-14T14:12:19.370Z - Hash: 5324e16e86ec61c0 - Time: 33145ms

2023-06-14T14:12:19.596 [Remote Cache]: storeFile /tmp/ddt-web-cache/cache_55629ea85dd76b8f9238f68141c2a3d3844f3440
ce60a98637a78f1e1b55d74a.tar.gz
------------------------------------------------------------------------------
Stored output to remote cache: nx-remotecache-filesystem
File: 55629ea85dd76b8f9238f68141c2a3d3844f3440ce60a98637a78f1e1b55d74a.tar.gz
------------------------------------------------------------------------------


 >  NX   Successfully ran target build for project scenario-tester (37s)
```

Then, similarly to before, if we run the build again, it is retrieved from the cache — however, this time rather than retrieving from the Nx internal cache, it checks the cache in the shared directory, sees that the file exists, then retrieves the file, and the outputs are restored.



```
bash-3.2$ nx build scenario-tester --prod --runner=ci
2023-06-14T14:32:58.273 [Remote Cache]: fileExists /tmp/ddt-web-cache/cache_55629ea85dd76b8f9238f68141c2a3d3844f344
0ce60a98637a78f1e1b55d74a.tar.gz true
2023-06-14T14:32:58.417 [Remote Cache]: retrieveFile /tmp/ddt-web-cache/cache_55629ea85dd76b8f9238f68141c2a3d3844f3
440ce60a98637a78f1e1b55d74a.tar.gz
------------------------------------------------------------------------------
Remote cache hit: nx-remotecache-filesystem
File: 55629ea85dd76b8f9238f68141c2a3d3844f3440ce60a98637a78f1e1b55d74a.tar.gz
------------------------------------------------------------------------------

> nx run scenario-tester:build:production  [remote cache]


Initial Chunk Files            | Names                                 |  Raw Size | Estimated Transfer Size
main.ba4217838d7ba59f.js       | main                                  |   8.07 MB |               1.45 MB
scripts.471162f81963ac03.js    | scripts                               | 507.52 kB |             142.98 kB
styles.e7ed0e0c81ba67ad.css    | styles                                | 397.52 kB |              33.79 kB
polyfills.56a487605c07b0af.js  | polyfills                             |  88.23 kB |              27.47 kB
runtime.bc79f0e49ce87d57.js    | runtime                               |   2.86 kB |               1.36 kB

                               | Initial Total                         |   9.04 MB |               1.65 MB

Lazy Chunk Files               | Names                                 |  Raw Size | Estimated Transfer Size
588.324d93e01248f0ce.js        | reallocate-view-reallocate-view-module |  30.41 kB |               4.43 kB

Build at: 2023-06-14T14:19:40.633Z - Hash: e931ea3f5eaf2506 - Time: 32765ms

 ————————————————————————————————————————————————————————————————————————————

 >  NX   Successfully ran target build for project scenario-tester (502ms)

    Nx read the output from the cache instead of running the command for 1 out of 1 tasks.
```

Impact on build and test times

So, to recap, we have now implemented Nx caching and configured the project to use a network fileshare as a distributed cache. Let's see what this does to our build times.

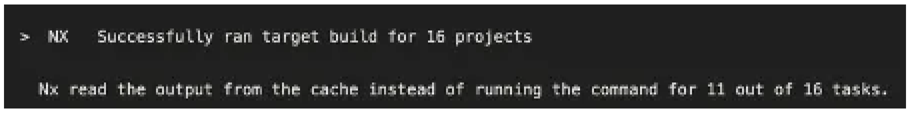
First, to get a baseline, let's take a pipeline where we run all build and test tasks from scratch (i.e., all apps will be built, and tests run for all apps and libs).

```
Running target 'build' for 16 projects: 15:05
```

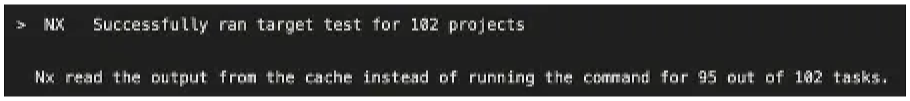
```
Running target 'test' for 102 projects: 17:47
```

Note: This is actually pretty quick by our standards — when running for all projects, full builds can often take 30 minutes, and full tests can take 40 minutes or *more depending on how busy the build boxes are.*

Let's compare this with a pipeline for a small pull request that only changes some of the affected projects, meaning some results can be retrieved from the cache.



```
>  NX   Successfully ran target build for 16 projects

   Nx read the output from the cache instead of running the command for 11 out of 16 tasks.
```



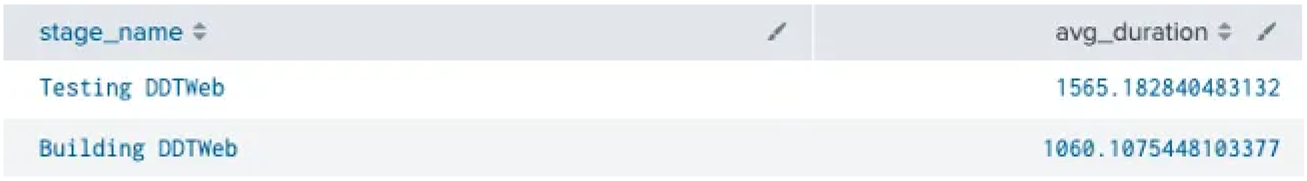
```
>  NX   Successfully ran target test for 102 projects

   Nx read the output from the cache instead of running the command for 95 out of 102 tasks.
```

Here we can see that for the build tasks, five of the 16 applications needed to be built from scratch, and for test tasks, only seven of the 102 projects needed the tests to be run.

`Running target 'build' for 16 projects [11 read from cache]: 6:08` **-8:57**

`Running target 'test' for 102 projects [95 read from cache]: 8:04` **-9:43**

If we dig further into the pipeline telemetry, we can work out precisely how much our times have improved. First let's see the average duration of our build and test steps for a few months before we enabled the caching:

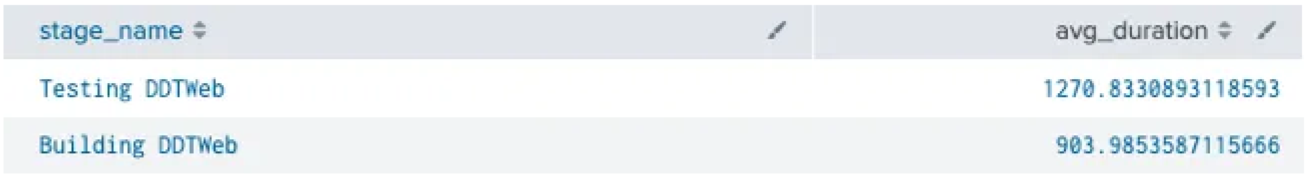


stage_name	avg_duration
Testing DDTWeb	1565.182840483132
Building DDTWeb	1060.1075448103377

`Building DDTWeb: 17:40`

`Testing DDTWeb: 26:05`

Now let’s find the average duration since enabling the caching:



stage_name	avg_duration
Testing DDTWeb	1270.8330893118593
Building DDTWeb	903.9853587115666

`Building DDTWeb: 15:03` **-2:37**

`Testing DDTWeb: 21:10` **-4:55**

In a real-world scenario the average improvement, while not inconsiderable, is not quite as dramatic as we first hoped.

We have a couple of possible explanations for this:

- *These durations are averages across both quality and publish pipelines. Anecdotally we have found that once a pull request has completed a quality pipeline and been merged, the subsequent publish pipeline completes much quicker because it can simply retrieve everything from the cache.*
- *In an average pipeline, the Nx task execution to build and test our projects is a fairly small amount of the total pipeline duration, which includes external factors such as resolving dependencies, downloading from pipeline caches, and uploading test coverage that all take roughly the same time in each pipeline.*

However, the time we save does start to add up. Our team merges on average about eight pull requests a day. Each of those will run at least two quality pipelines (usually more) and a publish pipeline. So, for an average day, assuming each PR results in three builds, we can estimate the following:

Average time saved per PR: **22:36**

Average time saved per day: **3:00:48**

Average time saved per working week: **15:04:00**

Conclusion

With some simple changes to our monorepo setup, implementing Nx distributed caching meant we were able to make some significant improvements to our team's build and test times.

Not only are we saving on average over 15 hours a week waiting for tasks to complete (and therefore reducing the time to get our pull requests merged and changes deployed), but it's 15 hours of computation time saved — and precious resources freed for builds for other teams.

. . .

Learn more about technology careers at BlackRock.

Monorepo Nx Build Cache Fintech

Published in BlackRock Engineering

868 followers · Last published Apr 10, 2025

Follow

Learn how we're solving complex engineering problems at BlackRock.

Written by BlackRockEngineering

Follow

1.2K followers · 9 following

Official BlackRock Engineering Blog. From the designers & developers of industry-leading platform Aladdin®. Important disclosures: http://bit.ly/17XHCyc

No responses yet

Piyali Das

What are your thoughts?

More from BlackRockEngineering and BlackRock Engineering

In BlackRock Engineeri... by BlackRockEngineeri...

Unravelling the micro-frontends puzzle with contract testing

In BlackRock Engineeri... by BlackRockEngineeri...

Open Sourcing the Aladdin SDK: Empowering Python Developers...

Thomas describes some of the unique challenges that arise when building micro…

Sep 28, 2023 16

By: Vedant Naik, Lead Engineer, Aladdin Studio, and Eli Kalish, Product Manager II,…

May 7, 2024 56

In BlackRock Engineeri… by BlackRockEngineeri…

The BlackRock Messaging System

This article gives you a glimpse into how we've implemented and scaled the…

Dec 4, 2018 61 1

In BlackRock Engineeri… by BlackRockEngineeri…

Domain-Driven Asset Management

Alan Moore describes the use of Domain-Driven Design for Asset Management at…

Mar 1, 2023 67 2

See all from BlackRockEngineering

See all from BlackRock Engineering

Recommended from Medium

In Stackademic by mikeleppanen

Track Your Finances with Leptos & Rust—Part 1: Introduction and…

Welcome to the first part of our series, where we'll build a complete finance tracking…

6d ago 52 1

Sohail Saifi

Kubernetes Is Dead: Why Tech Giants Are Secretly Moving to…

I still remember that strange silence in the meeting room. Our CTO had just announced…

Jun 7 2.5K 103

Sidharrth Mahadevan

Nx Monorepo Essentials: Integrating a NestJS Backend (Pa…

Welcome to the final installment of our Nx Monorepo Series! If you've been following…

Feb 13

In Utopian by Derick David

Cursor Is Doomed

Bloomberg's 'fastest growing startup ever has an expiration date

Jun 17 552 47

Deepak Sharma

iOS 26 Just Left Flutter Devs Behind

And No One's Talking About It — Until Now

Jun 10 716 31

In Coding Nexus by Civil Learning

Top 7 MCP Servers for AI-Driven Development

A couple of years ago, I was interested in a Next.js project, manually copy-pasting SQL...

Jun 18 174 1

See more recommendations