Course Hero

Theia Cache Reheating

Connor Clark

Version 1.1, 2018-02-12

Table of Contents

1. Overview

This document details the process by which `Theia` will handle automated cache reheating, without any window for cache-misses.

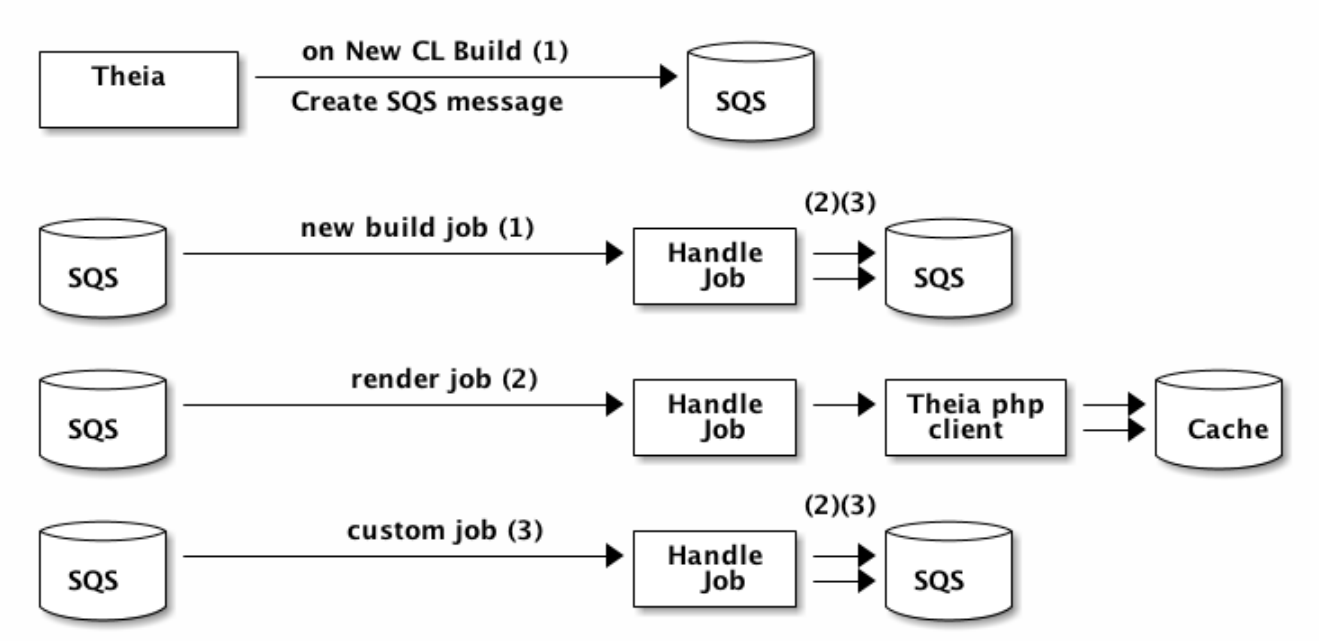
Following a new build of a Component Library, the rendering results stored in cache need to be reheated (read: updated). Without this step, render results from the old build will continue being served from cache, until it expires (and a request then makes its way to `Theia`). Alternatively, the entire cache could be cleared following a new CL build, which would result in many cache-misses for the first users requesting a particular rendering result.

We suggest using AWS SQS, and wish to handle a couple orders of magnitude more than `Theia` 's only current use case, `study-guides`, which is on the scale of 1000s of render results in cache.

2. Background

Reheating the cache is a process that has been handled manually for the Course Hero literature study guides. Following a push, someone would have to manually run a command to reheat the items in cache, involving coordination w/ DevOps. This process is being designed with automation in mind, which will reduce the overhead involved with a code push and enable a more seamless continuous deployment.

3. Details



1

```
{
  "MessageAttributes": {
    "Type": "new-build-job",
    "ComponentLibrary": "..."
  }
  "Body": {
    "builtAt": "...",
    "commitHash": "...",
    etc.
  }
}
```

2

```
{
  "MessageAttributes": {
    "Type": "render-job",
    "ComponentLibrary": "...",
    "Component": "..."
  }
  "Body": {
    ...props
  }
}
```

3

```
{
  "MessageAttributes": {
    "Type": "producer-job",
    "ComponentLibrary": "..."
  }
  "Body": {
    ...CL-specific data...
  }
}
```

The reheating process begins with a `new-build-job`. The message attributes will designate which Component Library was built. The job handler will delegate handling of this initial job message to callbacks, which should be registered for each specific Component Library.

In the simplest scenario, the initial job will queue up one `render-job` for each render request to process. However, a more granular approach should be used if any of the following are true:

1. There are many items that need to be reheated. Creating all the jobs could be a large task in itself.
2. Fault tolerance is deemed to be important.

(1) Should be true for most (if not all) use cases. The large number of jobs to create encourages a pagination approach. For example, instead of creating 1,000,000 `render-jobs` from `new-build-job`, create 1,000 `producer-job` s, where each job is responsible for creating 1,000 `render-job` s.

(2) Should also be true for most cases. If a job were to fail (perhaps because of a temporary network issue), the work that needs to be redone should be minimal. Breaking up the work into many jobs helps with this.

Example 1: `study-guides` CL

`new-build-job` will determine all the published course study guides. For each one, a `producer-job` will be created, each with a message attribute specifying the CL (and thus, which callback handler to use), and a message body specifiying a course. It will also create a single `render-job` for the index page.

`producer-job` will fetch the relevant course data, and for each page (CourseBlock, SubtopicBlock, and SectionBlock), a `render-job` will be created*.

*there is actually a limit to how big a SQS message can be, and unfortunately, the props needed for the `study-guides` CL vastly exceeds this limit. So, the above is an ideal implemention. The actual implementation will not create a job for each page. Instead, the `study-guides` handler for `producer-job` will issue render requests to Theia directly. The props needed for the index page is much smaller, so that can still be done with a `render-job`.

Example 2: hypothetical `documents` CL

`new-build-job` will determine the max document id. For each slice of 100 documents, a `producer-job` will be created, each with a message attribute specifying the CL (and thus, which callback handler to use), and a message body specifiying the range of document indicies to handle.

`producer-job` will enqueue a `render-job` for each document within its indice slice.

4. Open Questions

1. How to handle the removal of old items from the cache? Ex: a document that existed at one point, but was removed and the corresponding Theia cache render result is no longer needed.

A: 2 possible solutions:

a. Set a large-ish TTL for all cache items (1 year). Basically a punt.

b. This first thing `new-build-job` does could be to set a TTL value for all existing cache items for the CL. If a job ends up writing a value for an existing cache key, remove this TTL value. At the end of the reheating process, any values that still have a TTL set should be safe to remove. Allow the TTL process to do the actual removal. TTL should be configured to be a duration strictly larger than how long it should take to run all the jobs.

Option b is the suggested route to take. If we introduce a new column `producer_group` to the DynamoDB cache, updating the cache for a single producer group (all items that a single `producer-`

`job` creates) would work like this (using `study-guides` as an example):

1. A previously published study guide is updated and republished.
2. A single new `producer-job` is created (with `producer_group` set to the course name), which will be configured to prune the cache as laid out in option b, but only for items in the cache whose `producer_group` is a match.
3. TTL eventually kicks in and remove old items associated with previous version of study guide

5. Task Breakdown

1. Theia should emit an SQS `new-build-job` when a new build is finished
2. Monolith should have a TheiaJobProcessCommand. It should offer an easy interface to create new jobs and register CL-specific callbacks for jobs. Use an interface to implement CL-specific job handling.
3. Create ECS scaling task

6. Discussion

Direct all discussion to this Slack thread.