Improve performance with caching

November 10, 2022 • Jan Cerman • 13 min read

Make your apps more responsive, resilient, and generally perform better with caching. Caching helps reduce the wait time that you and the users of your app face when your app gets content from external resources like APIs.

Key points

- A cache is an extra layer that stores responses from external resources like Kontent.ai APIs.
- For non-demanding apps where temporarily outdated content is fine, use time-based caching.
- For apps where up-to-date content is critical, synchronize changes using webhooks or Sync API.

Cache in a nutshell

A cache is an additional layer of storage used to quickly retrieve data in your app. With Kontent.ai, cache sits between your app and an API. The cache contains entries with responses from the API.

When your app fetches content, it first checks whether the content is already in the cache. If it's available in the cache, there's no need to contact an API. If it's not yet cached, your app makes an API request and stores the response in its cache. When the content is updated in Kontent.ai, the cached response needs to be invalidated.

Because specific parts of your content might depend on other parts, you need to keep track of the relations among the cached responses, i.e., their dependencies. When your app invalidates a cached response, it should also invalidate any other related cached responses.

Cached responses and their dependencies on Kontent.ai objects.

See the diagram on https://viewer.diagrams.net?lightbox=1&nav=1#Uhttps%3A%2F%2Fraw.githubusercontent.com%2FKenticoDocs%2Fkontent-docs-diagrams%2Fmaster%2Fdevelopment%2Fcache-dependencies.drawio

Choose your cache strategy

Your caching strategy depends on your attitude towards stale content. Stale content is the one that has expired in cache. This means the content in Kontent.ai had been updated but your app continues to serve stale content.

If you're fine with stale content being outdated for 15 minutes or an hour, use **time-based caching**. This way you cache each response for the same amount of time.

If stale content is an issue and you want to ensure your cache is always up to date, you need to **synchronize content changes** based on recent content changes in your Kontent.ai project.

Fine with stale content? Use time-based caching

For client-side apps and basic mobile apps, we recommend using time-based caching. This means invalidating cached responses after a specific amount of time.

1. It's easier to implement.
2. It performs better because your app doesn't need to know about recent content changes and process them.

The expiration time should align with your users' expectations and their tolerance to stale content.

To improve your app's performance, set shorter expiration times for the types of content that change often, like articles, and longer times for static content, like navigation. With this approach, your app gets content directly from Delivery API.

This approach is also suitable when you don't have the resources to maintain a server-side app or proxy server to synchronize recent changes.

Need latest content? Synchronize content changes

If you're building mobile or web apps where up-to-date content is critical and you cannot afford stale content, you need to synchronize latest content changes. You can either receive and process webhook notifications, or fetch recent changes at your pace with Sync API.

Both webhooks and Sync API require a server-side app on your side. This can be a proxy server for your mobile app, a backend for your client-side app, or a regular web app.

- With webhooks, your app needs to be always available so that it can receive notifications at any time.
- With Sync API, your app regularly synchronizes changes in intervals of your choice.

Whenever your app is notified about a content change (either via a webhook notification or by asking Sync API), your app needs to invalidate specific items in its cache and fetch their latest version from Delivery API.

Your app needs to identify dependencies in the responses and invalidate cache entries based on the information it gets either from the webhook notifications or Sync API.

Combine content synchronization with time-based caching

For demanding mobile apps and client-side apps, we recommend you use time-based caching for the content your get from your server-side app.

This way you keep your client-server connections low (fewer requests for content) while keeping the cache invalidation logic on the server-side app.

Example: Cache implementation 101

Let's go through the general process of storing API responses in your app's cache.

The following sample scenario assumes a web application that powers a blog site. The site can show a list of blog posts, display a specific blog post, and organize posts by tags. With Kontent.ai, each blog post is a content item and each tag is a taxonomy term.

Step 1: Define your app's business layer

It's a good practice to define specific custom actions within your app (such as retrieving a blog post) and rely on those custom actions in your app. Think about the actions that your app performs regularly and create methods for them.

Using the blog site example, these actions can be:

- Get a blog post
- Get a list of blog posts
- Get a list of blog posts by category

In pseudocode, the actions might look like the following:

JavaScript

```
// Without custom actions, you use the Delivery client directly, specifying its parameters each time.
Client = DeliveryClient("<YOUR_PROJECT_ID>")
response = Client.items()
    .type("<type_codename>")
    .limit(10)
    .skip(20)

// With custom actions, you use named action to perform business logic.
// GetPost action retrieves a blog post by its codename.
GetPost("<post_codename>")
// GetPosts action retrieves a list of blog posts and provides paging.
GetPosts( <skip>, <limit>)
// GetPostsByCategory action retrieves a list of blog posts tagged with a specific category, and provides paging.
GetPostsByCategory("<category_codename>", <skip>, <limit>)
```

One of the benefits of using your own custom actions is that you can use a combination of the action's name and input parameters to name the API response (that is compose a cache entry key) and store it in the cache.

Step 2: Implement logic for caching your content

When you retrieve a blog post using your custom `GetPost()` action, you get a JSON response with the specified blog post. Whenever you get a response from the API, you store the response in the cache.

To cache the response, create a cache entry that uses a naming pattern such as `<action_name>|<action_parameters>` for its key. Use this pattern to uniquely identify the responses within the cache for each of your custom actions.

For example, if you retrieve a blog post named *My blog post*, you name the cache entry key for the response `GetPost|my_blog_post`. You can adjust the pattern to suit your needs and naming conventions.

Once you put the response in the cache, you're done. The next time your app calls `GetPost("my_blog_post")`, it retrieves the blog post from the cache without having to make any request to the API.

Handle cache dependencies

If fresh content is important to you, correct cache dependency handling is crucial. Let's go through how you can identify the cache dependencies and store them in cache.

Imagine your app requests a blog post and gets an API response with several components and a few links to content items. The linked items are dependencies of the blog post. If the dependencies change, so should the blog post.

The structure of the API response looks similar to the simplified JSON below.

- In the `item` object, you find a content item representing the blog post itself.
- In the `modular_content` object property, you find the components and linked items as separate object properties.

You need to go through the API response, differentiate content items from components, and add the content items as dependencies for the cached JSON response.

To uniquely identify the cache dependencies, use a naming pattern such as `<object_type>:<object_codename>`. Using the content item from the simplified JSON, the dependency name would be `item:guest_blog_post`. This way, you can invalidate the correct dependencies whenever there's a change in your content items.

JSON

```
{
    "item": {
        "system": {
            "id": "f4b3fc05-e988-4dae-9ac1-a94aba566474",
            "name": "My blog post",
            "codename": "my_blog_post",
            "language": "default",
```

```
            "type": "blog_post",
            "sitemap_locations": [],
            "last_modified": "2022-10-20T12:03:48.4628352Z"
        },
        "elements": { ... }
    },
    "modular_content": {
        "guest_blog_post": {
            "system": { ... },
            "elements": { ... }
        },
        "n2dfcbed2_d7a1_0183_4324_a2282f735f48": {
            "system": { ... },
            "elements": { ... }
        }
    }
}
```

Differentiate content items from components

When you get content items from Delivery API, the `modular_content` object in the API response contains both content items and components. Structurally, components and content items look the same.

To tell content items and components apart, find the object's ID and check the **third group** of characters in the ID.

- If the characters do NOT start with 01, for example, ce0288e7-294c-**46e5**-b9bc-b086656d5c48, it's a content item.
- If the characters start with 01, for example, ce0288e7-294c-**01e5**-b9bc-b086656d5c48, it's a component.

Guidelines on creating cache dependencies

Use the following guidelines for constructing your app's cache dependency logic.

For a Delivery API response with:

- **Single content item**
  - If the number of objects in the `modular_content` object property is below 30, add cache dependencies for all the objects.
  - If the `modular_content` property contains too many objects to define as separate dependencies, add a special general dependency for any content item. In such case, your app invalidates the cached response whenever any other content item is invalidated.
- **List of content items** – Add cache dependencies for all the content items in the list.
- **Single taxonomy group** – Add a cache dependency for the taxonomy group object returned in the response.
- **List of taxonomy groups** – Add a cache dependency for all the taxonomy groups in the list.

Once you're done adding the logic, you need to specify when each dependency should be invalidated.

Guidelines on invalidating cache dependencies

Use the following guidelines to specify which cache dependencies must be invalidated after your app receives a webhook notification or fetches delta updates from Sync API.

Cache invalidation with webhooks

When your app receives a webhook notification, it needs to process the notification and invalidate cache dependencies based on the type of content that changed.

- **Content item** – Pair the content item in the notification with its cache dependencies and invalidate the dependencies. Also, invalidate cached lists of content items.
  - For example, if you're caching paged responses, the removal of one content item from the first response would affect all the following paged responses.
- **Content type** – Invalidate all cache dependencies of content items based on the type.
- **Taxonomy group** – Pair the taxonomy group with its cache dependencies, such as content items tagged with the terms from the group, and invalidate them. Also, invalidate cached lists of any objects. Taxonomy groups may be used in any content item and content type, you need to refresh any cached lists of content items and content types.

For example, if you receive a notification with an object in the `items` array and that object's codename is `guest_blog_post`, you invalidate a cache dependency identified as `item:guest_blog_post`. This in turn invalidates any JSON responses dependent on the modified content item.

JSON

```
{
  "data": {
    "items": [
      {
        "id": "e5d575fe-9608-4523-a07d-e32d780bf92a",
        "codename": "guest_blog_post",
        "language": "default",
        "type": "blog_post"
      }
    ],
    "taxonomies": [
      {
        "id": "4794dde6-f700-4a5d-b0dc-9ae16dcfc73d",
        "codename": "tags"
      }
    ]
  },
  "message": { ... }

```

```
    }
}
```

Cache invalidation with Sync API

When your app synchronizes recent changes using Sync API, it needs to process the delta updates it gets and invalidate cache dependencies for the modified content items.

For example, if you get a delta update about a change in a content with the codename `guest_blog_post`, you invalidate a cache dependency identified as `item:guest_blog_post`. This in turn invalidates any JSON responses dependent on the modified content item.

JSON

```json
{
  "items": [
    {
      "codename": "guest_blog_post",
      "id": "7adfb82a-1386-4228-bcc2-45073a0355f6",
      "type": "blog_post",
      "language": "default",
      "collection": "default",
      "change_type": "changed",
      "timestamp": "2022-10-06T08:38:40.0088127Z"
    }
  ]
}
```

What's next?

- Caching content for mobile apps helps you avoid hitting the Delivery API fair use limits.
- Use offline caching with Gatsby to take your website user experience one step further.
- Know the differences between the linked content and components in the Delivery API.
- Get the right content by filtering content items.
- Map your content types to strongly typed models in your app for a better development experience.