Strategies and Solutions for Enhancing Frontend Image Processing in TYPO3

① Trigger Page Rendering After Each Content Change

In this strategy, page content is re-rendered whenever modifications are made in the backend. Ideally, only the modified content elements would trigger their own re-render, whether it's a single element edit or a mass edit, managed by \TYPO3\CMS\Backend\Controller\EditDocumentController.

A key consideration in this strategy involves deciding on frontend cache behavior — determining whether caches are flushed with each content modification or if they remain until manually cleared by an editor. This decision might involve various TCEMAIN.* (clearCacheCmd, clearCache_disable, clearCache_pageGrandParent, clearCache_pageSiblingChildren - more details here) configuration options for precise cache management. Additionally, ensuring the page renders in the modified language is essential.

Currently, by default, modifying page content automatically updates it on the frontend, leading to an automatic cache refresh.

This solution should be pretty straightforward to put in place, but it's not cut out for handling the trickier cases. It's mainly meant for making changes directly through the Page module, so it falls short when edits are made through the List module or any special backend modules that developers might add to their extensions. Plus, a single record could show up on multiple pages, either because of plugins or the Insert Record Content Element, and tracking down every single one of those pages to refresh their cache could be a real headache. It might not even be wise to try refreshing the content for many pages all at once. Also, this approach doesn’t really work for content that gets updated on the fly, like paginated data on the frontend, or different views (list, grid) with images, which can be toggled by the users.

Workspaces as an Integral Part of TYPO3 by Default

An additional concept related to this process involves separating the re-rendering of page content from the current frontend content and its cache. This would necessitate integrating workspaces into the core content management workflow, rather than offering it as an optional

extension. Thus, by default, without specific workspace or stage configurations, users would operate within and publish to the Live workspace. However, any modifications to content or pages would require pressing the 'Publish' button. This setup allows users to edit multiple content elements or pages and choose whether to discard or publish these changes. In this scenario, there exists a pre-live state for modified content that could also initiate the previously mentioned content re-rendering, ensuring that processed images are generated. Such features - regarding content publishing - are implemented in Neos CMS or Drupal.

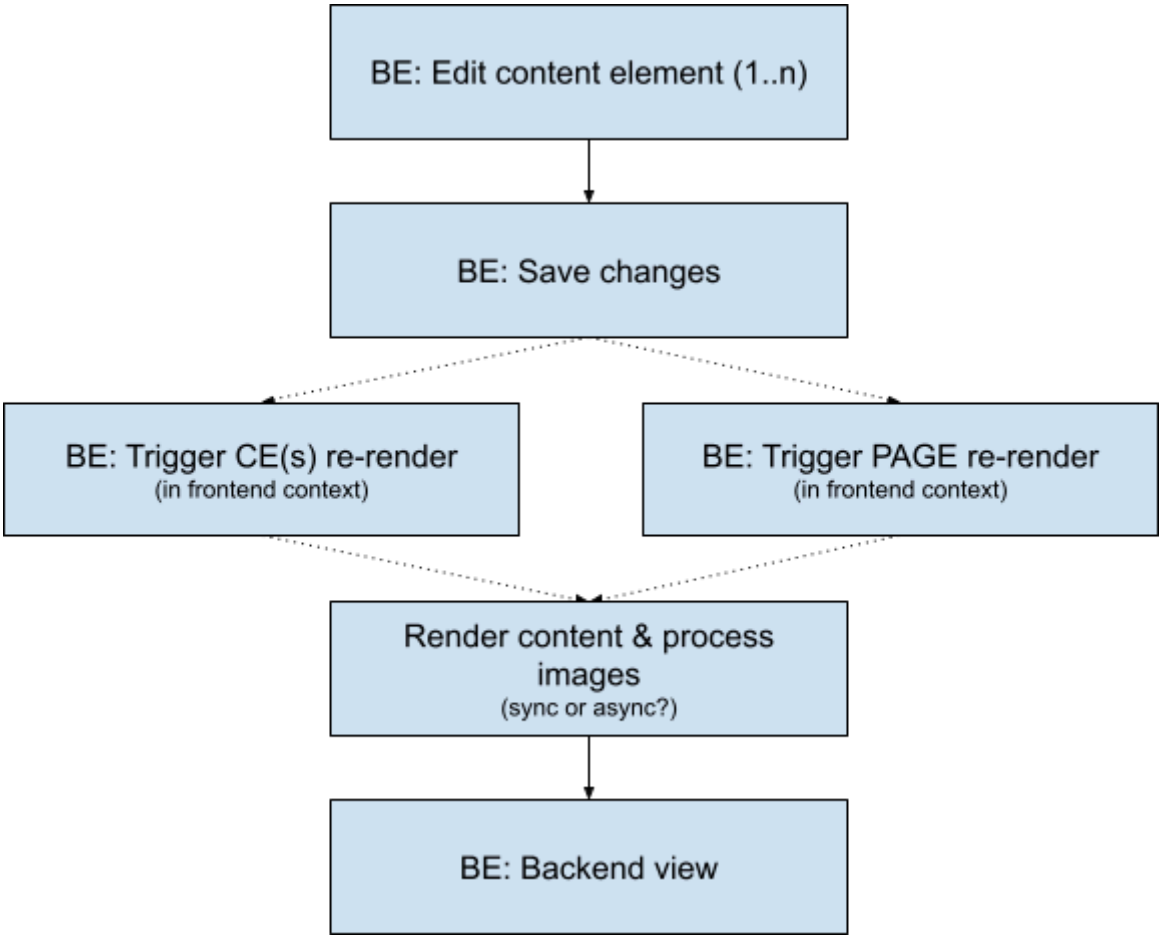


Diagram: General flow for content re-rendering triggered from the backend

> **There is no any good solution to recognize and render only content elements with images (either on main or relational record)?**

> **Custom TypoScript configuration can perform additional image processing on content elements (CE) or pages, and this processing will not be triggered when only a single CE is re-rendered**

Pros	Cons
- Image files will be processed immediately following any content changes saved in the backend. - Only the images that need processing will be handled (these already processed will not block page rendering) - Approach where only edited content elements are re-rendered, makes the whole process faster - Approach where the whole page is re-rendered might be slower but still the result would be the same - having processed image files when serving page on the frontend - Having page pre-cached will make the page load faster later on	- Needs to be triggered each time the changes in content are made - Might be a bit complicated setup if we take into account caches and languages - This solution may not be effective for all setups. On websites where paginated data is rendered or views like grid/list are toggled, we often won't be able to initiate image processing for pages other than the current one or for views other than the default. While this approach might accelerate page rendering for the default view, it doesn't apply to content loaded asynchronously, which often requires user interaction - Some records, such as News, can be rendered across multiple pages (1..n) using or displaying images in various configurations. This variability makes it challenging to ensure that all images are processed after the record has been updated - Custom record types, such as News, are typically managed through a dedicated module or from the List View and rendered by specific plugins. It will not be easy to determine on which details page a given news record is rendered or which list it appears on; therefore, triggering image re-processing on all these pages will not be straightforward - There might be rendering configurations for certain data, such as *RSS Feeds*, defined exclusively through TypoScript, and it won't be straightforward to locate these configurations and initiate their execution for image processing after changes have been made to the content on some pages - A little more resources used as the rendering will take place more often, though the rendered content should be lighter in general

■ - real pros / cons
■ - non real pros / cons

Usage on shared or self hosted server

- ☑ Is this a good solution for shared hosting (see challenges)?
- ☑ Is this a good solution for self managed servers (VPS, dedicated server etc.)?

Will it work for more complex setups (paginated data etc.) and when the Page module is not used?

- ☐ Yes (read more)

Enable toggling on/off per environment

- ☑ Yes, it should be possible (read more)

② Generate and Return Temporary URLs for Deferred Image Processing

This approach logic would be similar to the one that is currently implemented for the backend through the **DeferredBackendImageProcessor**. There should be a dedicated processor which will generate temporary URLs for not yet processed images that need some processing. Such a temporary URL would point to a dedicated route, which will point to an endpoint where the file processing would take place (this way page rendering will not be blocked by images processing).

A **locking strategy** must be implemented to prevent redundant processing by multiple processes targeting the same file. However, this approach may have drawbacks during high-traffic periods, especially when numerous large images require processing. Each lock could delay many requests, potentially leading to resource limitations, such as reaching the maximum number of PHP processes on shared hosting environments etc.

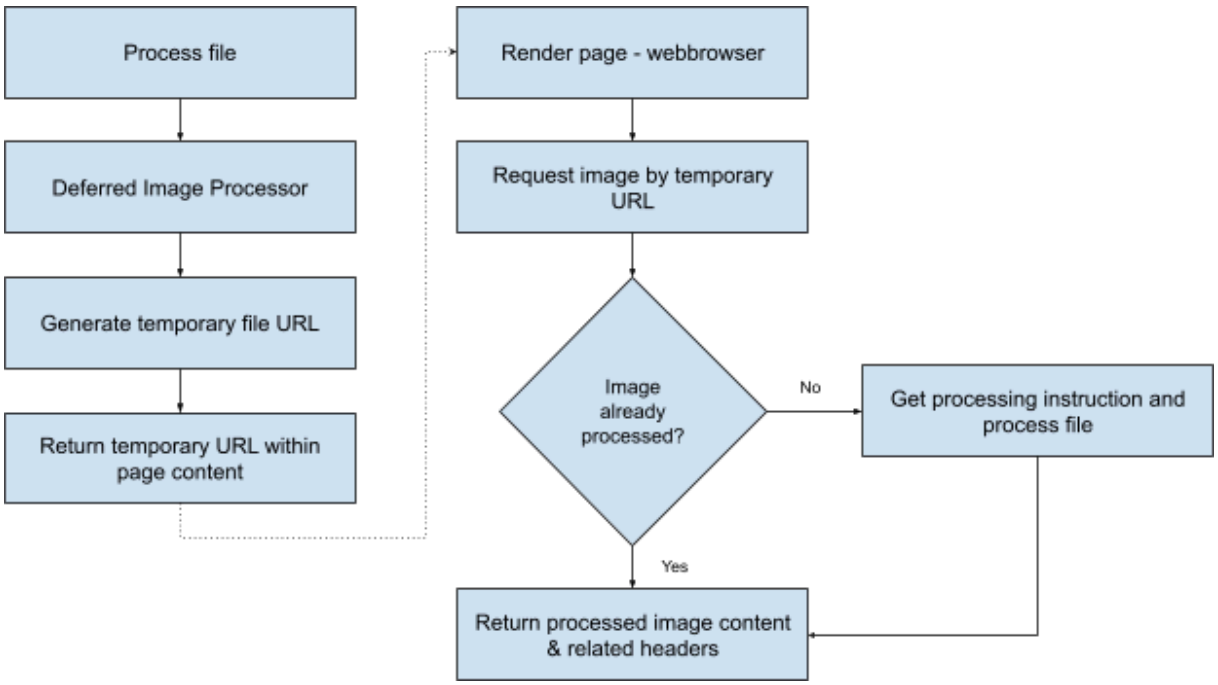
For a setups when there are multiple application instances (load balancer is used) the processed images and the locking mechanism must be shared to avoid processing the same image on each app instance.

The processing should be triggered as early as possible by **dedicated middleware** to make the whole request as lightweight as possible.

An important security aspect to handle, involves the dedicated endpoint for deferred image processing. It's essential to restrict processing capabilities to prevent unauthorized use. Specifically, **it should be verified that any image requested for processing is legitimately part of the page content**, rather than being sourced from arbitrary requests by attackers. Such a measure is crucial to mitigate the risk of DDoS attacks triggered by the exploitation of the image processing feature. This could be done by using some hash/salt verified by the backend.

Temporary URLs should not be cached, meaning the sections of the page where they appear stay uncached until the final URLs are generated and used. Additionally, there's a potential SEO concern if a search engine indexing robot is the first to access the page, encountering content with temporary URLs.

Another option could be to set the page as uncached (bypass page caching) when it contains unprocessed images. However, this approach might significantly impact performance on high-traffic websites.



Is there any good solution to recognize and render only content elements with images (either on main or relational record)?

Pros	Cons
- No need for additional configuration for the WEB server (.htaccess, nginx config) to process the request - By deferring the image processing to the point where an image is actually requested, the initial page load time can be significantly reduced. This is especially beneficial for pages with a lot of images that do not need to be displayed immediately - This approach can help in spreading out the server load over time. Since images are only processed as needed, the server does not have to process all images at once, which can be beneficial during peak traffic times - Resources are used more efficiently as only the images that are needed are processed. This can lead to savings in storage space and processing power, especially if there are many images that end up never being requested by users.	- Managing cache effectively can be more challenging with this approach. Ensuring that images are properly cached after the first request. The temporary URLs should not be cached, so probably this part of the page should stay uncached until final URL is generated - Using locking strategy might result with pending requests and it might result with some limits reached (like PHP processes on shared hosting etc.). The locking strategy must be shared across multiple instances of the same app (behind load balancer etc.) - Requires additional SELECT / UPDATE queries to DB to fetch file processing configuration to be used - For applications behind load balancers, especially those running multiple instances with identical code, it's essential to have shared processed files and a locking mechanism. This ensures the avoidance of redundant image processing - The first time an image is requested, users may experience a delay while the server processes the image. This could negatively impact the user

experience, especially if the processing takes longer than expected - but page processing / loading won't be blocked after the caches are flushed

■ - real pros / cons
■ - non real pros / cons

Usage on shared or self hosted server

- ☑ Is this a good solution for shared hosting (see challenges)?
- ☑ Is this a good solution for self managed servers (VPS, dedicated server etc.)?

Will it work for more complex setups (paginated data etc.) and when the Page module is not used?

- ☑ Yes (read more)

Enable toggling on/off per environment

- ☑ Yes, it should be possible (read more)

③ Generate Final URLs for Images Without Processing During Page Rendering

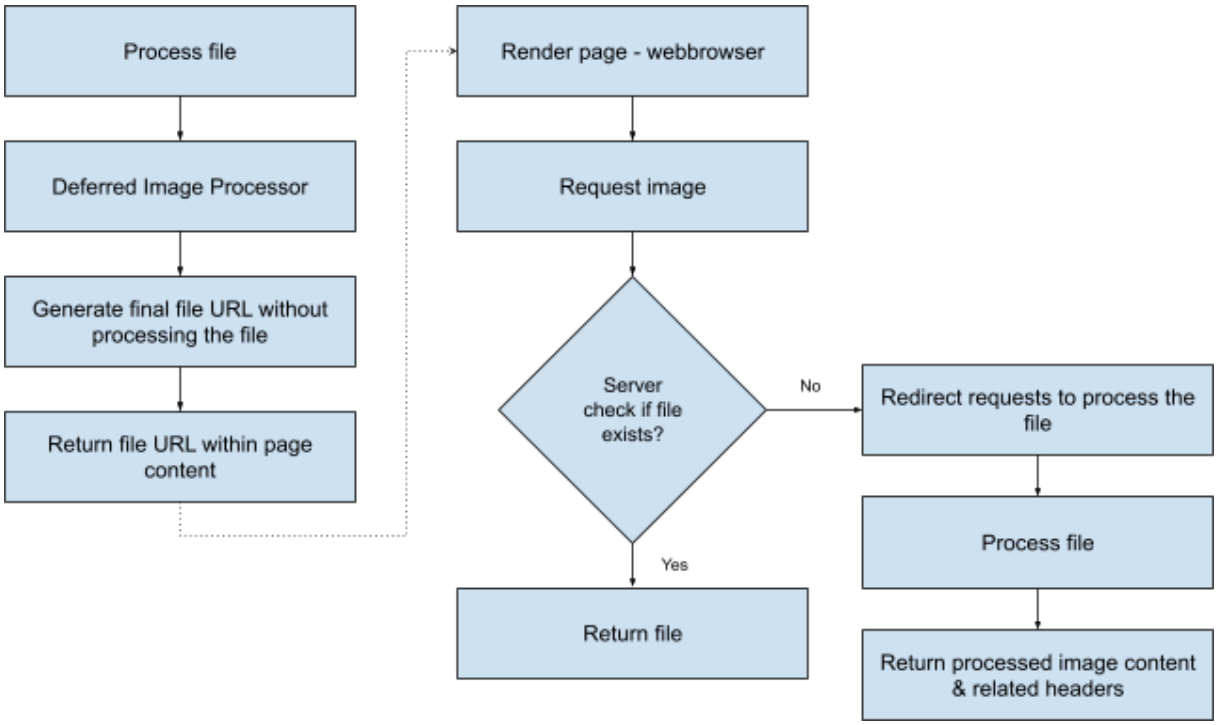
This approach, likely superior to its predecessor, ensures the final URL for a processed image is generated even when processing is deferred. Consequently, concerns related to caching and SEO are mitigated when generating page content.

It necessitates a specialized file processor to store processing instructions within the database (specifically in the *sys_file_processedfile* table) without initiating processing immediately, yet still allowing the retrieval of the final URL for the processed file. This URL is then returned and embedded within the rendered page content.

Upon a web browser's request for a specific image, the server checks whether the file exists at the specified path. If it exists, the file is served directly; if not, the request is redirected to a dedicated route or the index.php file and managed by specialized middleware. This will require some additional configuration on the server.

A similar solution is available through the ext:deferred-image-processing extension. This extension verifies the existence of the requested file and, if absent, processes the request via dedicated middleware. It relies on processing instructions stored in the dedicated database table *tx_deferredimageprocessing_file*, retrieving processing details using the file's public URL. Additionally, this table can serve as a queue, allowing images to be processed in bulk based on the records within, through a specific command executed by the Scheduler. This functionality

proves beneficial for pages that are infrequently visited, ensuring images on these pages are processed in advance by this command.



Pros	Cons
- By deferring the processing of images and serving final URLs instantly, the page content can be rendered faster, enhancing the user experience - The file processing is handled through the dedicated route/middleware which directly returns processed files, so no other page content is rendered which could slow down the whole process/request - There are not problems with caching or SEO in context of page content delivered by the backend	- Requires additional yet simple configuration for the web server. The TYPO3 documentation would provide such configuration but it would still require the developer to have and access to Apache or Nginx configuration file - Requires additional SELECT / UPDATE queries to DB to fetch file processing configuration to be used - A locking strategy is required to prevent multiple requests from processing the same file simultaneously. The lock must be released in cases where image processing fails, for instance, due to insufficient disk space for the processed file. - Using locking strategy might result with pending requests and it might result with some limits reached (like PHP processes on shared hosting etc.)

■ - real pros / cons
■ - non real pros / cons

Usage on shared or self hosted server

- ☑ Is this a good solution for shared hosting (see challenges)?
- ☑ Is this a good solution for self managed servers (VPS, dedicated server etc.)?

Will it work for more complex setups (paginated data etc.) and when the Page module is not used?

- ☐ Yes (read more)

Enable toggling on/off per environment

- ☑ Yes, it should be possible (read more)

④ Delegate Image Processing to External Service

This approach does not require or trigger any file processing by TYPO3 (except when GIFBUILDER was used). In this scenario each entry point that can trigger file processing, like view helpers (*MediaViewHelper*, *ImageViewHelper*, *Uri\ImageViewHelper*), TypoScript objects *IMAGE* and *IMG_RESOURCE* or using the file processing API by developers directly in their extensions, should receive the URL pointing to the Image Proxy server with the original file path and all the parameters describing what processing should be done on such file.

There should be a dedicated file processor (?) which will handle the processing task and only return the URL pointing to the Image Proxy. All the processing is being mage on the Image Proxy side and we do not care (from TYPO3 page rendering perspective) how long the images will be processed etc. because we do not block the rendering content here.

> **The structure and path of the final URL, including whether it points to public or private files**
> **and whether the Image Proxy can access them, are topics for further discussion.**

```
None
https://image-proxy.com/storage/path/filename.png?crop=1&w=200&h=100
```

Example URL generated by the file processor. Image Proxy should know how to load the file and process it. It might be different for different implementations

In this solution, we are exploring the option of using an Image Proxy service, which can be managed and maintained by our developers. They would have complete control over its configuration. Additionally, we are considering a paid option where control and configuration might be more limited, and the maintenance would rely on third-party companies and their employees.

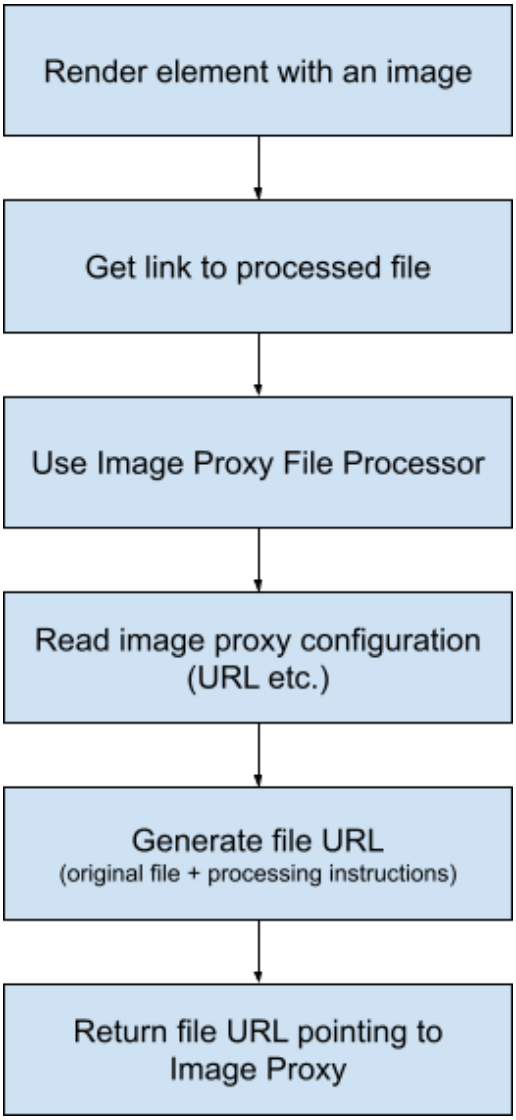


Diagram: The general workflow of using an Image Proxy for file processing

Pros	Cons
- Offloading heavy image processing tasks to an image proxy, allows TYPO3 ro focus on delivering content, potentially leading to faster page load times and a smoother user experience - Does not store processed files locally (no need for an extra space) - Image proxies can dynamically scale to handle high loads of image processing requests, making it easier to manage traffic spikes without additional configuration or resources on CMS	- GIFBUILDER has to be handled locally - Dependency on external services - relying on an external service (if it is not managed by our devs) introduces a point of failure outside of our control. If the image proxy service experiences downtime or performance issues, it could affect the website and presented content - While some image proxy services offer free tiers, more extensive usage or advanced features often come at a cost. These costs can accumulate, especially for

- Many image proxies offer advanced processing options like smart cropping, automatic format selection, and compression that might not be readily available or easy to implement
- Good solution for shared hostings where the resources might be limited (max requests, PHP processes, U/O limits etc.)
- Serving optimized images can significantly reduce bandwidth consumption, which is particularly beneficial for mobile users and can lead to cost savings on data transfer
- Offloading to an image proxy can reduce the risk of security vulnerabilities associated with image processing libraries directly on TYPO3 server

high-traffic websites. Moreover, even when such a service relies on our own infrastructure, it still demands additional resources and manpower to maintain, which, in turn, incurs additional costs

- When using third-party services (not hosted on our own infrastructure), the data privacy laws and regulations must be considered, especially if we handle sensitive or personal images
- Integrating an image proxy may require additional configuration and ongoing maintenance), especially if some custom or complex image processing is needed. Here TYPO3 could provide just an entry point to plug-in Image Proxy handling and the concrete implementation could/should rely on dedicated extensions etc.
- No information about processed file in the TYPO3 database table *sys_file_processedfile* (but we probably do not need it at all)

■ - real pros / cons
■ - non real pros / cons

Usage on shared or self hosted server

- ☑ Is this a good solution for shared hosting (see challenges)?
- ☑ Is this a good solution for self managed servers (VPS, dedicated server etc.)?

Will it work for more complex setups (paginated data etc.) and when the Page module is not used?

- ☑ Yes (read more)

Enable toggling on/off per environment

- ☑ Yes, it should be possible (read more)

⑤ Use Placeholder Images and Process Images Through Scheduler Task

In this solution a processor designed to assign a placeholder image's URL (whether a generic image, a low-resolution bitmap, or an SVG representation) along with a scheduled task that processes images in a background queue should be used.

Whenever image processing is initiated, the processor will return the URL of a placeholder image if the processed file is not yet available. It's crucial that these placeholder images remain uncached within the page content cache. This ensures that subsequent requests can return the URL of the processed image instead of the placeholder.

Placeholder images should be provided by default; however, users should have the option to specify their own files, such as those in local storage or within an extension.

This solution is similar to the one that Drupal uses for thumbnail rendering on the backend side, where selected media types can have/use queued thumbnail rendering.

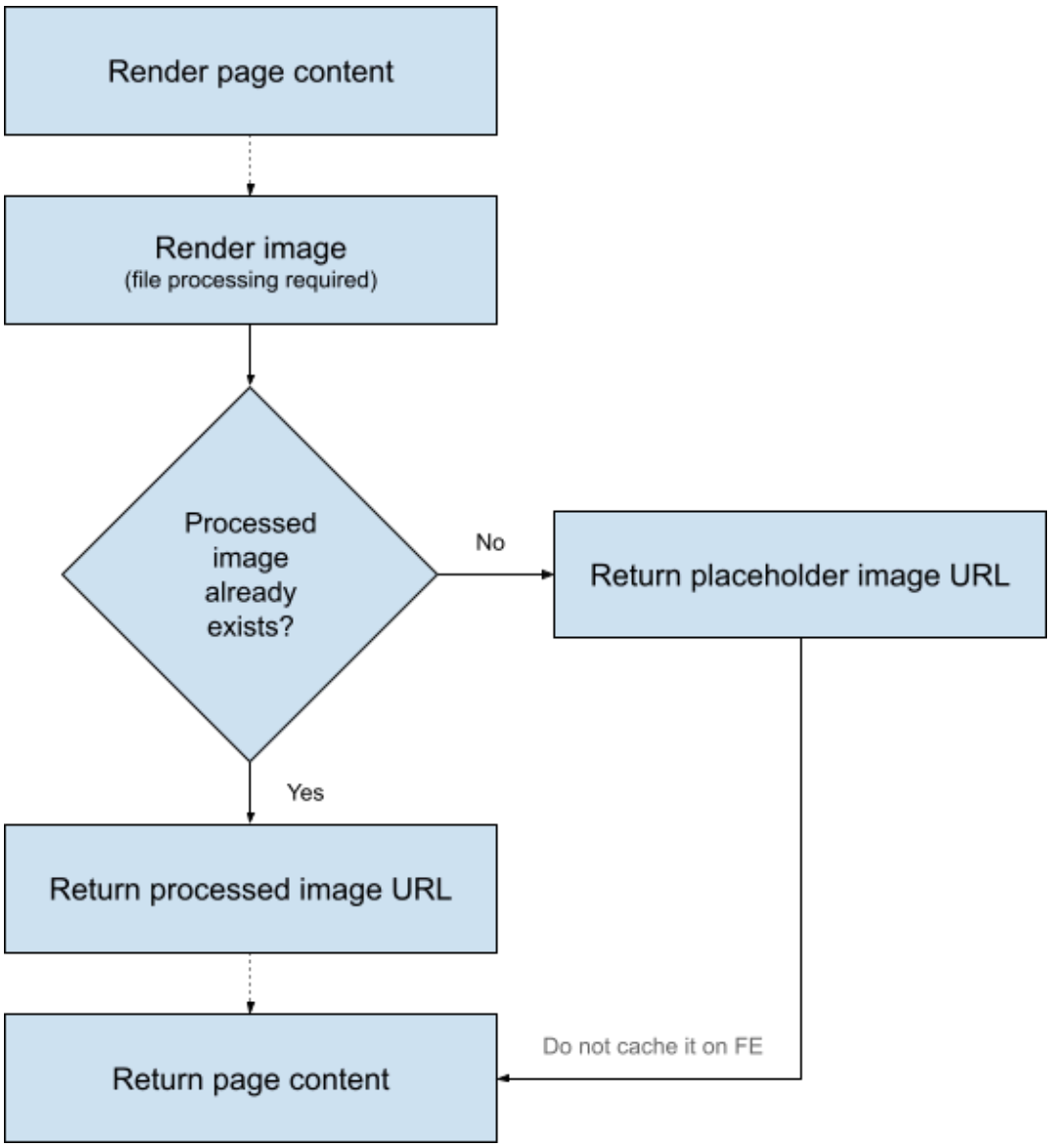


Diagram: The general workflow when using placeholder images for unprocessed files

Pros	Cons
- Page content rendering is not blocked by images processing	- Frondend renders the placeholder image until the page is refreshed and the processed files already exist (their

- Placeholder images are rendered with target file size/dimensions, so they appear in the page content the same as the target files would
- Placeholder images could be configurable for each site / environment to match some design concepts etc.

processing was triggered according to the queue)

- If the queue with images to process will grow up for some reason, there might be significant delay until all images get processed
- If for some reason queue fails to work or the Scheduler task hangs, page content might end-up for a longer time with placeholder images
- Placeholder images should not be cached with page content, only processed images should be part of fully cached page
- There will be situations where some of the images have been processed and are displayed on the page, while others still have placeholders.

■ - real pros / cons
■ - non real pros / cons

Usage on shared or self hosted server

- ☑ Is this a good solution for shared hosting (see challenges)?
- ☑ Is this a good solution for self managed servers (VPS, dedicated server etc.)?

Will it work for more complex setups (paginated data etc.) and when the Page module is not used?

- ☐ Yes (read more)

Enable toggling on/off per environment

- ☑ Yes, it should be possible (read more)

Implementing Multiple Solutions with Options to Toggle Them On/Off

In the TYPO3 code, it's possible to consider and decide that not just a single, but multiple solutions can be implemented. Each solution could be toggled on or off through configuration settings that vary by environment. This flexibility allows for different approaches to be adopted in testing or production environments.

A critical question arises regarding whether developers or administrators should be allowed to activate multiple solutions at once. Often, such combinations might not be practical. For instance, activating page rendering after each content change does not align with using an image proxy, as file processing would be outsourced to an external service in any case.

Another combination that warrants a closer look involves activating page rendering after every content change while also enabling deferred image processing. The latter generates final URLs for processed files without initiating the file processing itself.

If implementing multiple solutions is chosen, a decision must be made regarding whether it is advisable for developers or administrators to use one or multiple solutions simultaneously, or whether the TYPO3 core should restrict the use of more than one solution at a time.

> **Requires decision**

Challenges in Managing Complex Websites

TYPO3 is versatile enough to construct both simple and highly complex websites. The complexity may stem from employing paginated data or various view types, such as lists or grids, which users can toggle. Changing these views could initiate server-side image processing before the final content is delivered, potentially leading to increased waiting times and delays.

In considering the optimal solution for image processing and minimizing page rendering times (ensuring image processing does not hinder it), it's crucial to account for such scenarios and ensure that the chosen solution adequately manages them.

Comparing Solution Usage Across Various Entry Points

This table details the entry points (places) where image processing can be initiated during page content rendering. It details the first hit of the image processing process, creating a dedicated record in the *sys_file_processedfile* database table etc.

Solution	**Core ViewHelpers (used also by CE)**	**TypoScript**	**Custom extension and code**
	Image processing triggered through ViewHelpers, which are also used by core content elements	Image processing triggered from custom TS code	Custom plugins, content elements etc.
① Trigger Page Rendering After Each Content Change	ViewHelper can add extra processing instructions so it needs re-rendering each time the used file is modified (crop area changes etc.)	It is possible to define processing instructions for files (through IMAGE or IMG_RESOURCE objects etc.) so it needs rendering each time the used file is modified (crop area changes etc.)	File processing API usage gives a lot of flexibility so for custom plugins, content elements it needs re-rendering each time the used file is modified (crop area changes etc.)
② Generate and Return Temporary URLs for Deferred Backend Images	No file processing is triggered (processed file data stored in DB)	No file processing is triggered (processed file data stored in DB)	No file processing is triggered (processed file data stored in DB)
③ Generate Final URLs for Images Without Processing During Page Rendering	No file processing is triggered (processed file data stored in DB)	No file processing is triggered (processed file data stored in DB)	No file processing is triggered (if proper API is used) (processed file data stored in DB)

④ Delegate Image Processing to External Service	No file processing is triggered (no processed file data stored in DB)	No file processing is triggered (no processed file data stored in DB)	No file processing is triggered (if proper API is used) (no processed file data stored in DB)
⑤ Use Placeholder Images and Process Images Through Scheduler Task	No file processing is triggered (processed file data stored in DB)	No file processing is triggered (processed file data stored in DB)	No file processing is triggered (if proper API is used) (processed file data stored in DB)

Image Processing Challenges on Shared Hosting Platforms

PHP Process Execution Limits

Shared hosting often imposes limits on the number of PHP processes that can run simultaneously or the maximum execution time for a single script. Image processing tasks, especially those that are complex or involve multiple images, can be quite resource-intensive and may exceed these limits. When the maximum execution time or process limit is reached, the server may terminate the process, causing image processing tasks to fail.

Concurrent Request Limits

Shared hosting providers typically limit the number of concurrent requests to the server to prevent any single user from over-utilizing server resources. During periods of high traffic, or when multiple users request image-heavy pages at the same time, these limits might be hit. This can lead to slower response times or even service unavailability, affecting not just image processing but the overall user experience on the website.

I/O Limitations

On shared hosting, disk I/O (input/output operations) are often limited. This can slow down the process of reading and writing image files to disk, especially for image-heavy applications.

Execution Time Limits

Shared hosts often impose strict limits on script execution times to ensure fair resource use across all users. Image processing can be resource-intensive and time-consuming, leading to timeouts.
Memory Limitations - Image processing, particularly of large files, can be memory-intensive. Shared hosting plans may have restrictive memory limits, causing out-of-memory errors during processing.

Version Compatibility Issues

Shared hosts may not always offer the latest versions of image processing libraries (e.g., GD Library, ImageMagick) or PHP, which can limit the functionality or efficiency of your image processing.

Security Restrictions

For security reasons, some shared hosts restrict the use of certain functions and libraries that might be necessary for image processing tasks.

Documentation for Implementation Details

If it will be decided to implement any of the proposed solutions, also documentation (as part of the official TYPO3 documentation) should be created, detailing the recommended setup and best practices, such as when using multiple app instances.

The documentation should include a separate section for each solution, covering the following areas:

- Explanation of the solution
- Details of the required configuration
- Demonstration of sample setups with examples
- Guidelines on best practices

Summary

The meeting with Simon Praetorious was a source of valuable feedback regarding the initial propositions. He pointed out several issues with approach no ① Trigger Page Rendering After Each Content Change and they were documented inside the Pros/Cons summary. We also agreed that the approach no ② Generate and Return Temporary URLs for Deferred Image Processing might be problematic due to proper caching handling and performance.

We agreed that the solutions ③ Generate Final URLs for Images Without Processing During Page Rendering and ④ Delegate Image Processing to Image Proxy Service might the be best to use as they will not have negative impact on page rendering performance, instead they will improve it a lot amd they do not require any special caches handling.

The solution ③ Generate Final URLs for Images Without Processing During Page will require some additional configuration on the server, but these days is should not be a problem for developers, and ready to use configuration would be shipped with the TYPO3 source code and documented and explain in the official documentation.

The ④ Delegate Image Processing to Image Proxy Service would likely offer only interfaces and abstract classes for extension developers to utilize in providing concrete implementations, given the variety of external services/solutions that could be applied. As part of this effort, a separate extension with an implementation for an open-source solution like imgproxy could be developed.

The ⑤ Use Placeholder Images and Process Images Through Scheduler Task approach requires managing the frontend caches to ensure that these images are not cached, as they

would otherwise link to the placeholder files. Having uncached parts on the page could negatively impact both performance and user experience. Overall, this solution presents more cons than pros, and reliance on a scheduler might sometimes lead to situations where it fails and the images are not processed. Although this might be a rare occurrence, it should still be considered.

From our perspective, those two solutions are the best candidates for improving image processing and accelerating page rendering in TYPO3:

- ③ Generate Final URLs for Images Without Processing During Page
- ④ Delegate Image Processing to Image Proxy Service

The solution ① Trigger Page Rendering After Each Content Change could be taken into consideration, knowing that it won't work for all cases and more complex setups. Maybe as a optional one - if developers decide to activate it.

Notes

Webp image format support

https://docs.typo3.org/c/typo3/cms-core/main/en-us/Changelog/13.0/Feature-88537-WebPImageFormatSupportForImageProcessing.html

---

Insights into Current TYPO3 Processes and Implementations

The following sections detail the existing processes and solutions for file processing in TYPO3. This content is intended purely for informational purposes, providing an overview of the current state without proposing any changes or enhancements.

Entry Points for Image Rendering on Frontend and Backend

This diagram illustrates the 'entry points' through which image rendering and processing may be initiated in the context of both frontend and backend. Essentially, there are three types of such entry points: view helpers, TypoScript objects, and image processing from custom extensions through a dedicated API. Ultimately, all these paths lead to the `File::process()` method, which manages the file processing under the hood.

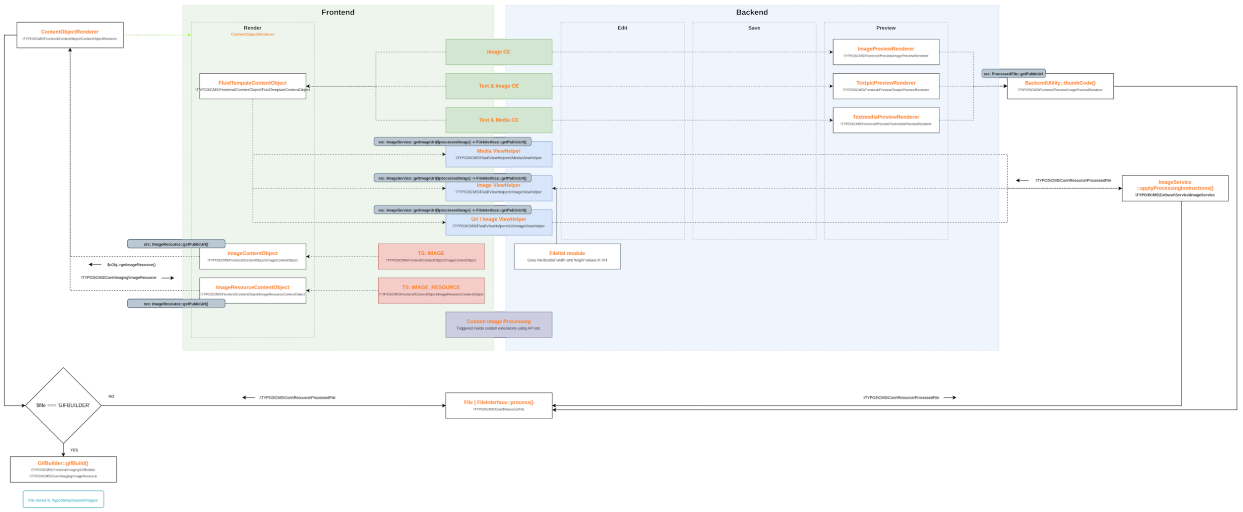


Diagram link:
https://drive.google.com/file/d/1s6L09okkKDtNVBCTgvtD_wWFISb_HloS/view?usp=sharing

Overview of Class Dependencies in Image Rendering

This diagram provides a general overview (though not exhaustive) of the dependencies among multiple classes involved in the image processing workflow. It highlights low-level classes that utilize libraries such as GD or Imagick, in addition to mentioning view helpers, content elements, and TypoScript objects.

The diagram highlights the 'API for Image Processing' layer, envisioned as a comprehensive set of interfaces for image manipulation, including cropping, rotating, resizing, and combining. This design allows for flexible underlying implementations and library usage to be adapted as necessary.

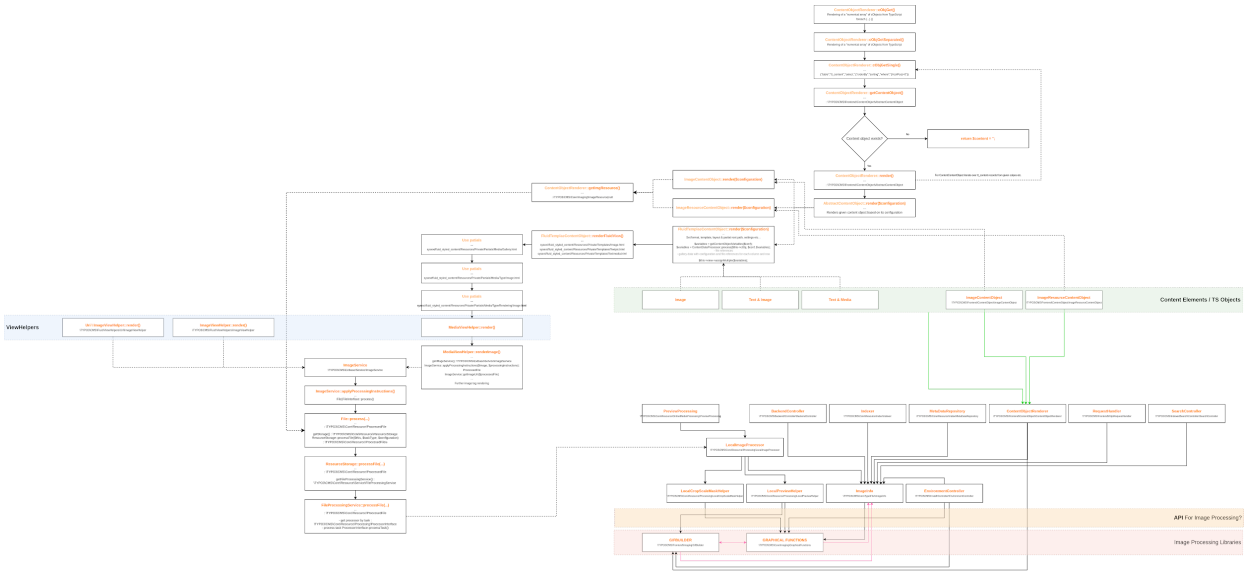


Diagram link:
https://drive.google.com/file/d/1HJlYXzgJ9XkeHlQVi6jeD4ux-LlMIe1Y/view?usp=sharing

Deferred Backend Image Processing

For the backend, an implementation for asynchronous thumbnails was introduced through the **DeferredBackendImageProcessor**, which, under the hood, generates a temporary URL for an image. This URL points to a backend route named **image_processing**, which has the following configuration:

```python
'image_processing' => [
  'path' => '/image/process',
  'target' => Controller\File\ImageProcessController::class . '::process',
],
```

The resulting temporary image URL generated will appear as follows:

```python
https://domain.com/typo3/image/process?token=8182a3bf143330ab7c91be2f422e5a1ba509d3bc&id=5
```

Here, the ***token*** parameter is used for security reasons, generated by the **FormProtectionFactory**, while the ***id*** parameter points to the corresponding record in the **sys_file_processedfile** table. This table stores all information about the processed file, such as the processing configuration, the relation to the original file, the temporary processing URL, etc.

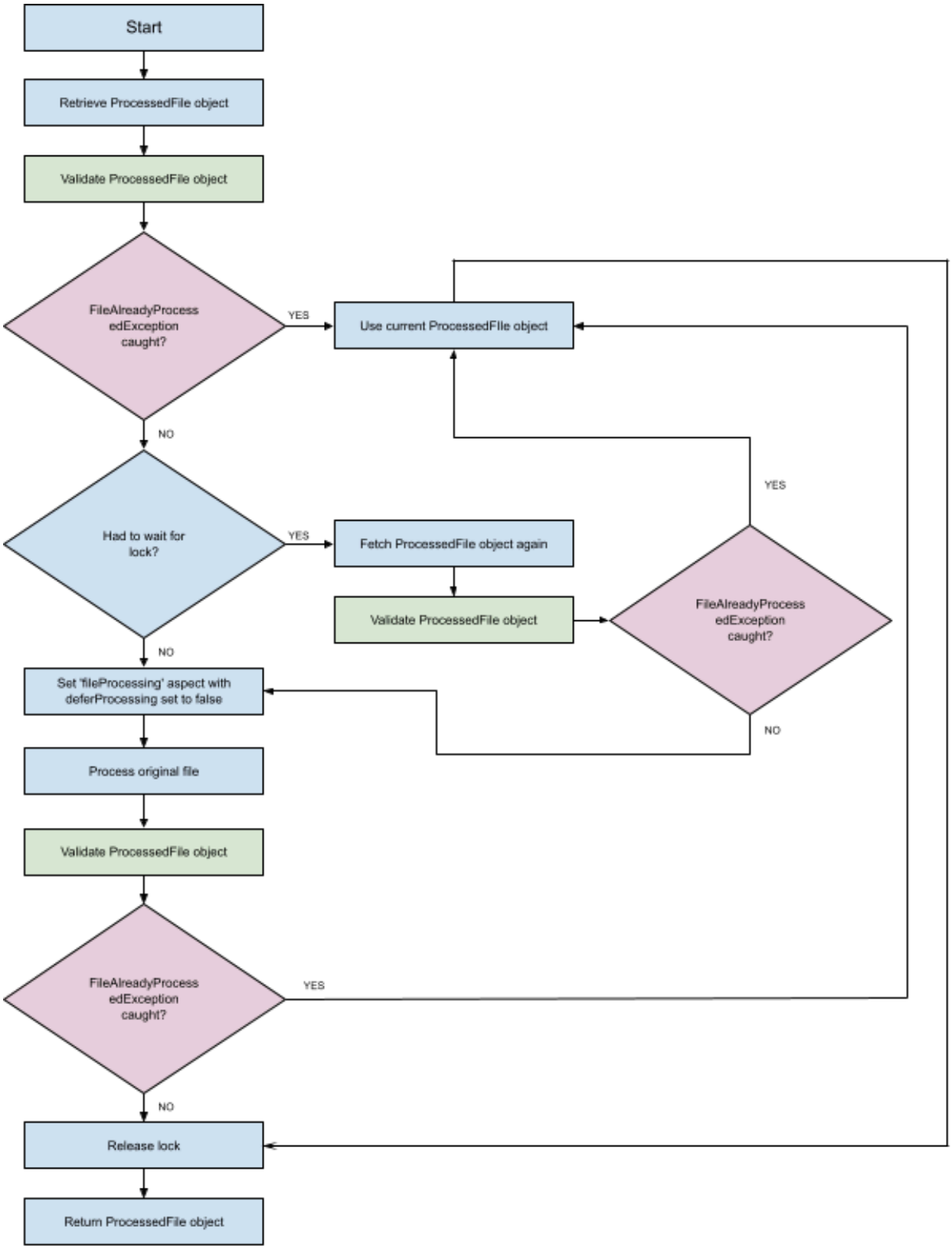
The **DeferredBackendImageProcessor** will execute the file processing task only if it meets multiple conditions, such as being in the backend request and the target file not existing yet, among others.

The **ImageProcessController** which handles the deferred image processing, internally uses **ImageProcessingService::process()** method which will:

- Retrieve the **ProcessedFile** object from the repository using the provided processed file ID (record from sys_file_processedfile table)
- Validate the **ProcessedFile** object to ensure it hasn't been processed yet
- Acquire a lock to prevent other processes from processing the same file simultaneously
- If the lock had to be waited for (another process acquired it), fetch the **ProcessedFile** object again and validate it to ensure it hasn't been processed yet

- Set the **fileProcessing** aspect of the **Context** with **deferredProcessing** set to false, indicating that deferred processing is disabled (this is checked inside **DeferredBackendImageProcessor::canProcessTask()** to determine if this processor should be used - not needed since the file must be processed now)
- Process the original file with the task identifier and processing configuration of the **ProcessedFile** object
- Validate the **ProcessedFile** object again after processing
- If a **FileAlreadyProcessedException** (thrown by the validation method if the file has already been processed) is caught, retrieve the **ProcessedFile** object from the exception
- Release the lock and return the **ProcessedFile** object

Described flow is illustrated on the following diagram:


Start
Retrieve ProcessedFile object
Validate ProcessedFile object
FileAlreadyProcessedException caught?
YES
Use current ProcessedFile object
NO
Had to wait for lock?
YES
Fetch ProcessedFile object again
Validate ProcessedFile object
FileAlreadyProcessedException caught?
YES
NO
Set 'fileProcessing' aspect with deferProcessing set to false
Process original file
Validate ProcessedFile object
FileAlreadyProcessedException caught?
YES
NO
Release lock
Return ProcessedFile object

Patches

- [FEATURE] Render thumbnails in file list module deferred
  This patch adds a new ViewHelper to render thumbnails deferred in the backend. This increases the performance of the file list.
- [BUGFIX] Implement deferred BE image processing consistently
  Change the implementation of backend deferred image processing to use a file processor, which is only but always used in the backend.

Processing files

When working with files, whether on the frontend or backend side, it is often required to process them. For example, with images, the thumbnails need to be rendered or cropped versions generated, etc. TYPO3 provides an easy-to-use API and ViewHelpers which, under the hood, execute the processing based on the provided configuration.

Through the Fluid templating engine it is possible to use those view helper to get desired image:

- `\TYPO3\CMS\Fluid\ViewHelpers\ImageViewHelper`
- `\TYPO3\CMS\Fluid\ViewHelpers\MediaViewHelper *`
- `\TYPO3\CMS\Fluid\ViewHelpers\Uri\ImageViewHelper`

They all use the **`\TYPO3\CMS\Extbase\Service\ImageService->applyProcessingInstructions($image, $processingInstructions)`** method to receive all the required that about processed file through the **`\TYPO3\CMS\Core\Resource\ProcessedFile`** object. However, this service internally operates on the original file object by calling the `\TYPO3\CMS\Core\Resource\File->process()` method, which expects two parameters:

- `string $taskType` (default: `ProcessedFile::CONTEXT_IMAGECROPSCALEMASK`)
- `array $configuration` (processing configuration, with crop area etc.)

* The \TYPO3\CMS\Fluid\ViewHelpers\MediaViewHelper can use a dedicated renderer if registered through \TYPO3\CMS\Core\Resource\Rendering\RendererRegistry.

```python
public function process(string $taskType, array $configuration): ProcessedFile
{
    return $this->getStorage()->processFile($this, $taskType, $configuration);
}
```

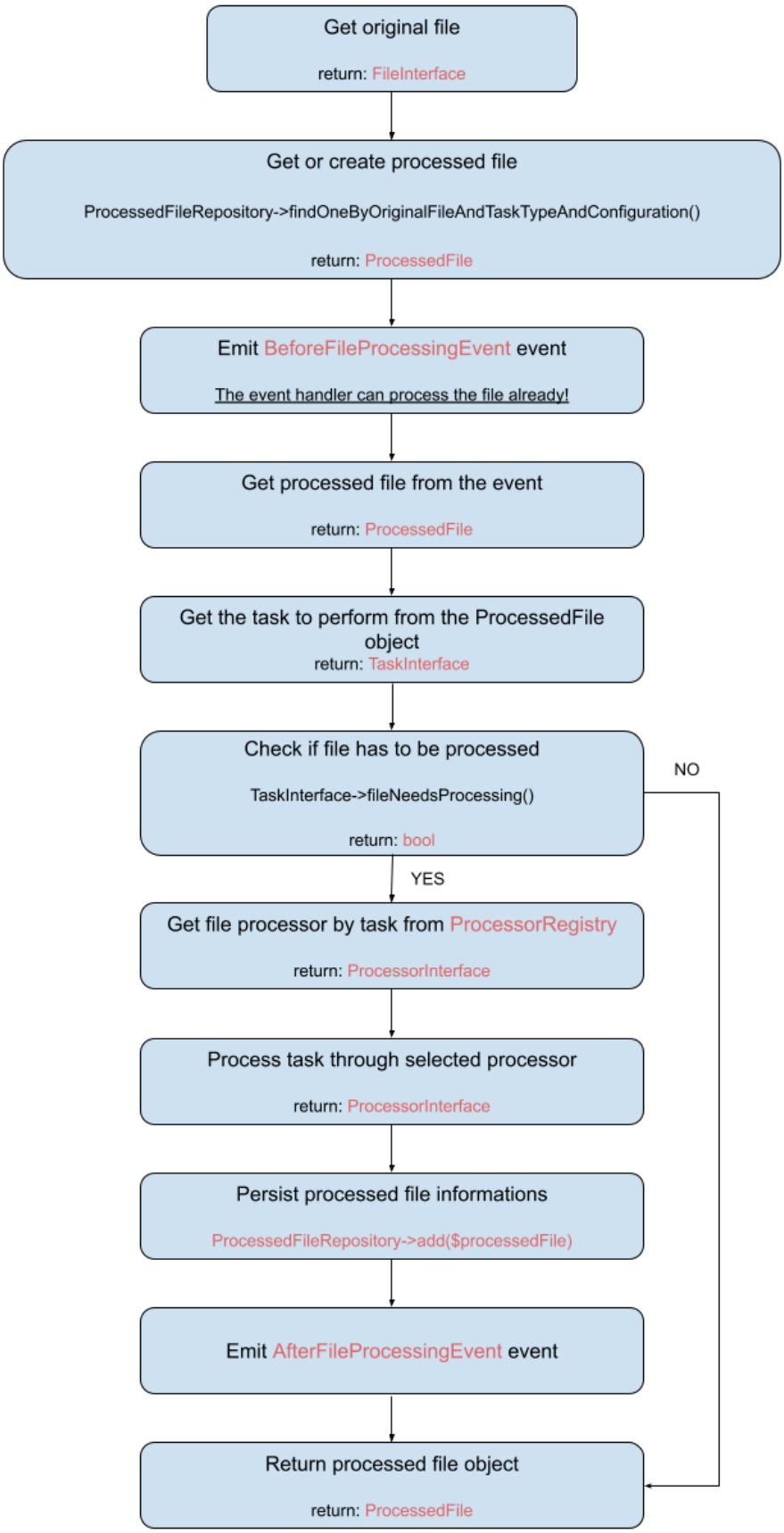
The File::process() function declaration

The complete sequence of classes and methods used to initiate file processing, starting from the `\TYPO3\CMS\Core\Resource\File->process()` method, is as follows:

```
Python
File->process(string $taskType, array $configuration)
  -> ResourceStorage->processFile(File $file, $taskType, array $configuration)
    -> FileProcessingService->processFile(File $file, $taskType, array $configuration)
```

Looking at this sequence, it is clear that the real heart of this task (file processing) is the **`\TYPO3\CMS\Core\Resource\Service\FileProcessingService`** class and its **`processFile`** function, which internally executes the following steps:

- Get the original **`File`** object
- Get or create a **`ProcessedFile`** object
- Emit the **`BeforeFileProcessingEvent`** event
- Retrieve the **`ProcessedFile`** again from the event (as it could have already processed the file)
- Identify the processing task (`TaskInterface`) to be performed
- Check if the processing task should be performed
- Search for an appropriate file processor (`FileProcessorInterface`) using ProcessorRegistry to perform the file processing task
- If needed, process the task with the selected processor
- Emit the `AfterFileProcessingEvent` event
- Return the `ProcessedFile`


Get original file
return: FileInterface
Get or create processed file
ProcessedFileRepository->findOneByOriginalFileAndTaskTypeAndConfiguration()
return: ProcessedFile
Emit BeforeFileProcessingEvent event
The event handler can process the file already!
Get processed file from the event
return: ProcessedFile
Get the task to perform from the ProcessedFile object
return: TaskInterface
Check if file has to be processed
TaskInterface->fileNeedsProcessing()
return: bool
NO
YES
Get file processor by task from ProcessorRegistry
return: ProcessorInterface
Process task through selected processor
return: ProcessorInterface
Persist processed file informations
ProcessedFileRepository->add($processedFile)
Emit AfterFileProcessingEvent event
Return processed file object
return: ProcessedFile

FileProcessingService steps execution inside the processFile(File|FileReference $fileObject, string $taskType, DriverInterface $driver, array $configuration): ProcessedFile function

The tasks and processors were often mentioned above. Tasks are small classes with properties like *name* and *type* (which define the task category, such as Image or Video, etc.). They hold information about the target file on which the task should be performed, the configuration that should be used to process the file, the target extension, and more. File processors, on the other hand, take on those tasks they can execute/handle. Based on the configuration provided by the task, they perform the file processing.

Processed file data is stored in the database table **`sys_file_processedfile`** and is used to populate the `ProcessedFile` object.

Column	Example value	Description
storage	*1*	The file storage identifier.
original	*3*	The original file's identifier.
identifier	*/_processed_/c/3/csm_test-image_c3bc652631.png*	The relative path of the processed file within the storage.
name	*csm_test-image_c3bc652631.png*	The name of the processed file.
processing_url		
configuration	*a:2:{s:5:"width";s:3:"a";s:6:"height";s:3:"32c";}*	Serialized configuration used for processing the file.
configurationsha1	*48ec22f851d7822181aeed9649f97929e5f0c410*	The SHA1 of the configuration used to process the file.
originalfilesha1	*d71acb0a98ecc24583bc403e3931e5f1e18e1aaf*	The original file's SHA1 (used to check if the original file has changed after processing).
task_type	*Image.CropScaleMask*	The type of task executed on the original file.
checksum	*ef57d2f3fc*	Unique checksum for the task's configuration, considering both the file and task type.
width	*32*	Width of the processed file.
height	*32*	Height of the processed file.

Structure and sample data for the sys_file_processedfile database table record

For use in the backend, there is a dedicated **`\TYPO3\CMS\Backend\ViewHelpers\ThumbnailViewHelper`** which generates an *img* tag with a special source URI for deferred thumbnail rendering. An example value for such a URI is: **`https://www.example.com/typo3/image/process?token=XYZ&id=1160`**. This approach ensures that page rendering isn't blocked until all thumbnails are rendered, but rather generates and fetches them asynchronously.

File Processing Tasks

When discussing file processing in TYPO3, it's essential to consider certain tasks that need to be executed on the file. These tasks themselves don't process the file; instead, they define through their configuration what should be done with the file. This job is then delegated to a file processor (which we will discuss in the next article of this series). The processor determines how and which tool to use for the task.

For each task, the name and a type can be assigned. The type specifies the task's category. In the TYPO3 core, there are tasks with the type set to **`Image`**, but other types can be considered like **`Video`**. For tasks that apply to multiple file types, the category would be **`General`**.

It is important to know on which file the task should be executed, so the task itself also contains information about the target file it relates to.

Essential Functions of File Processing Tasks

Each file processing task must implement the **`\TYPO3\CMS\Core\Resource\Processing\TaskInterface`**, which requires functions for:

- Retrieving the task name
- Retrieving the task type (category)
- Accessing the source file (`\TYPO3\CMS\Core\Resource\File`)
- Accessing the target file (`\TYPO3\CMS\Core\Resource\ProcessedFile`)
- Obtaining the processing configuration.
- Calculating the checksum of the processing configuration.
- Sanitizing the processing configuration
- Determining the target file name
- Identifying the target file extension
- Assessing the necessity for file processing
- Setting/checking the task's execution status
- Verifying successful execution of the task

TYPO3 also defines an abstract task through the **`\TYPO3\CMS\Core\Resource\Processing\AbstractTask`** class, which already declares some of the interface functions to facilitate and expedite the development of concrete tasks. Within this class, ready to use implementations for methods responsible for checking if a file needs

processing, obtaining the configuration checksum, sanitizing the configuration, and others are available.

Core Tasks for Image Processing

When it comes to tasks related to image processing, TYPO3 has two built-in tasks of the **`Image`** type, used across various parts of the code. These are:

CropScaleMask
`\TYPO3\CMS\Core\Resource\Processing\ImageCropScaleMaskTask`
A task that takes care of cropping, scaling and/or masking an image.

Preview
`\TYPO3\CMS\Core\Resource\Processing\ImagePreviewTask`
A task for generating an image preview

Since both tasks are of the **`Image`** type (category), they can only be executed by those file processors capable of handling this type of task. It's worth noting that processors are executed in a given order (according to their configuration). Once the first processor suitable for executing the task is found, it will handle the task, and no other processor will be checked or used.

Where TYPO3 Stores Information About Task Types

Information about all available file processing-related task types is stored in the global configuration array **`$GLOBALS['TYPO3_CONF_VARS']['SYS']['fal']['processingTaskTypes']`**.

```python
$GLOBALS['TYPO3_CONF_VARS']['SYS']['fal']['processingTaskTypes'] =
[
  'Image.Preview' =>
    \TYPO3\CMS\Core\Resource\Processing\ImagePreviewTask::class,
  'Image.CropScaleMask' =>
    \TYPO3\CMS\Core\Resource\Processing\ImageCropScaleMaskTask::class,
]
```

Configuration of the 2 default image processing tasks

In this array, the keys are formed by concatenating the task type and task name, separated by a dot, while the value points to the class that defines this task.

To easily find and obtain a task instance for a given task type, TYPO3 provides a special registry defined in the **`\TYPO3\CMS\Core\Resource\Processing\TaskTypeRegistry`** class. This registry (a singleton) is quite straightforward: it reads the configuration from the **`$GLOBALS['TYPO3_CONF_VARS']['SYS']['fal']['processingTaskTypes']`** array, stores it internally,

and exposes a function called **`getTaskForType`**. This function checks if a task with the given type is registered, and if it exists, it returns a new instance of it or throws an exception.

```
Python
$taskObject = $this->taskTypeRegistry->getTaskForType($taskType, $processedFile, $configuration);
```

Example for getting task of given type

Here's an important note: the task type passed as the first argument must match the key within the **`$GLOBALS['TYPO3_CONF_VARS']['SYS']['fal']['processingTaskTypes']`** configuration array, not the task type (category) defined by the task class itself. The names of the two task types defined and available in the TYPO3 core are also specified as constants within the **`\TYPO3\CMS\Core\Resource\ProcessedFile`** class:

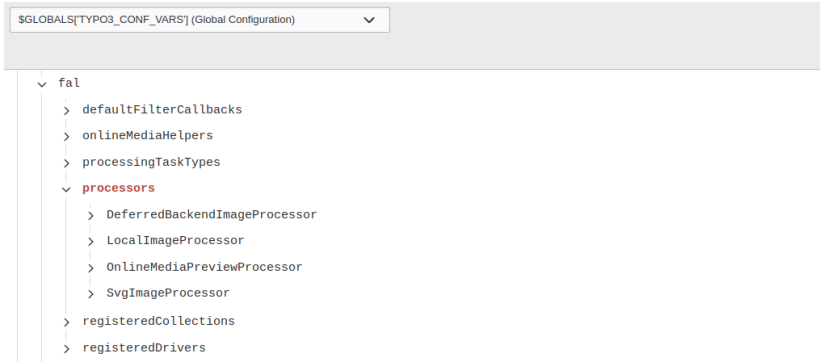
```
Python
\TYPO3\CMS\Core\Resource\ProcessedFile::CONTEXT_IMAGEPREVIEW = 'Image.Preview';
\TYPO3\CMS\Core\Resource\ProcessedFile::CONTEXT_IMAGECROPSCALEMASK = 'Image.CropScaleMask'
```

To get a task for one of these types, those constants can be used to pass the task type to the **`getTaskForType`** method of the **`\TYPO3\CMS\Core\Resource\Processing\TaskTypeRegistry`**

File Processors

A file processor is a class that can handle specified file processing tasks. The task provides instructions on what needs to be done with the file (for example, a task for an image file might include instructions to scale, crop, or rotate the file). However, how these actions are carried out and what tools are used to achieve them are the responsibility of the processor. This approach allows for instance, to change the tools or APIs used by the processor while keeping all the task declarations unchanged and valid at all times.

All the existing file processor configurations (either from the TYPO3 core or custom ones) are stored inside the $GLOBALS array:
`$GLOBALS['TYPO3_CONF_VARS']['SYS']['fal']['processors']`. This configuration can be easily checked through the **Configuration** backend module, which renders all data from the **`$GLOBALS['TYPO3_CONF_VARS']`** array.



Each file processor must implement the **`TYPO3\CMS\Core\Resource\Processing\ProcessorInterface`** interface, which mandates the definition of two important methods:

- **`canProcessTask`** - checks if processor can handle selected task
- **`processTask`** - performs the file processing task

It is important to remember that only one file processor will be used to handle any given file processing task, even if multiple file processors are defined. TYPO3 will iterate through the defined processors according to their configured order, and once the first one matches the conditions within its **`canProcessTask()`** function, this processor will be used to handle the file. Cascade file processing is not possible.

Python

```
public function canProcessTask(TaskInterface $task): bool
{
  return $task->getType() === 'Image'
    && in_array($task->getName(), ['Preview', 'CropScaleMask'], true)
    && empty($task->getConfiguration()['crop'])
    && $task->getTargetFileExtension() === 'svg';
}
```

Example check from the `SvgImageProcessor`, if it can handle selected task

List of all the TYPO3 file processors provided by default:

Processor name	Executed after	Executed before	Class

DeferredBackendImageProcessor	SvgImageProcessor	LocalImageProcessor OnlineMediaPreviewProcessor	TYPO3\CMS\Backend\Resource\Processing\DeferredBackendImageProcessor
LocalImageProcessor			TYPO3\CMS\Core\Resource\Processing\LocalImageProcessor
OnlineMediaPreviewProcessor	SvgImageProcessor	LocalImageProcessor	TYPO3\CMS\Core\Resource\OnlineMedia\Processing\PreviewProcessing
SvgImageProcessor		LocalImageProcessor	TYPO3\CMS\Core\Resource\Processing\SvgImageProcessor

Creating a Custom File Processor

Custom file processors in TYPO3 can be effectively registered through the **`ext_localconf.php`** file. This file plays a crucial role in the extension configuration process. The configuration should follow a specific format, as demonstrated in the example below:

```python
$GLOBALS['TYPO3_CONF_VARS']['SYS']['fal']['processors']['NewFileProcessor'] = [
    'className' => \MyVendor\ExtensionName\Resource\Processing\NewFileProcessor::class,
    'before' => ['LocalImageProcessor'],
    'after' => ['SvgImageProcessor'],
];
```

Example of the custom file processor registration

The *before* and *after* options allow for the definition of priority for the file processor, which is used by the **`ProcessorRegistry`** to determine the proper order. This is important because only a single file processor will be used to handle each file processing task – specifically, the first one that is identified through the registry.

The final step is to ensure that the defined processor class exists and implements the **`\TYPO3\CMS\Core\Resource\Processing\ProcessorInterface`** interface.

File Processor Registry

In TYPO3, there is a registry class called **`ProcessorRegistry`**, defined in the **`\TYPO3\CMS\Core\Resource\Processing\ProcessorRegistry`** class file. As the name suggests, this serves as the registry for all existing file processors.

When the registry is instantiated, it utilizes the **`\TYPO3\CMS\Core\Service\DependencyOrderingService`** to create an array of all the file processors registered through the configuration dedicated to FAL: **`$GLOBALS['TYPO3_CONF_VARS']['SYS']['fal']['processors']`**. It then sorts them by their configured order using the *before* and *after* options and stores this sorted array in an internal property for later use.

This registry exposes one public method, **`getProcessorByTask(TaskInterface $task): ProcessorInterface`**, which is responsible for finding a processor suited to perform the provided task. Internally, this method will loop through the configured file processors, instantiate them, and execute the **`canProcessTask()`** function on each, until it returns true for the first time. Then the loop is broken, and the current file processor is returned to handle the file processing task.

Documentation:

https://docs.typo3.org/m/typo3/reference-coreapi/main/en-us/ApiOverview/FileProcessing/Index.html

Rendering Image Files on the TYPO3 Frontend

In TYPO3, various media files, including images, can be utilized and rendered in multiple ways. Rendering an image or generating a URI for an image can be achieved through TypoScript with the **`IMAGE`** or **`IMG_RESOURCE`** objects. For operations within Fluid templates, several view helpers are available: **`\TYPO3\CMS\Fluid\ViewHelpers\ImageViewHelper`**, **`\TYPO3\CMS\Fluid\ViewHelpers\MediaViewHelper`**, and **`\TYPO3\CMS\Fluid\ViewHelpers\Uri\ImageViewHelper`**. These tools allow for the rendering of an *img* tag, the rendering of any media type through a specific renderer (when available), and the creation of a URI for an image, respectively.

Steps Executed Inside the ImageViewHelper

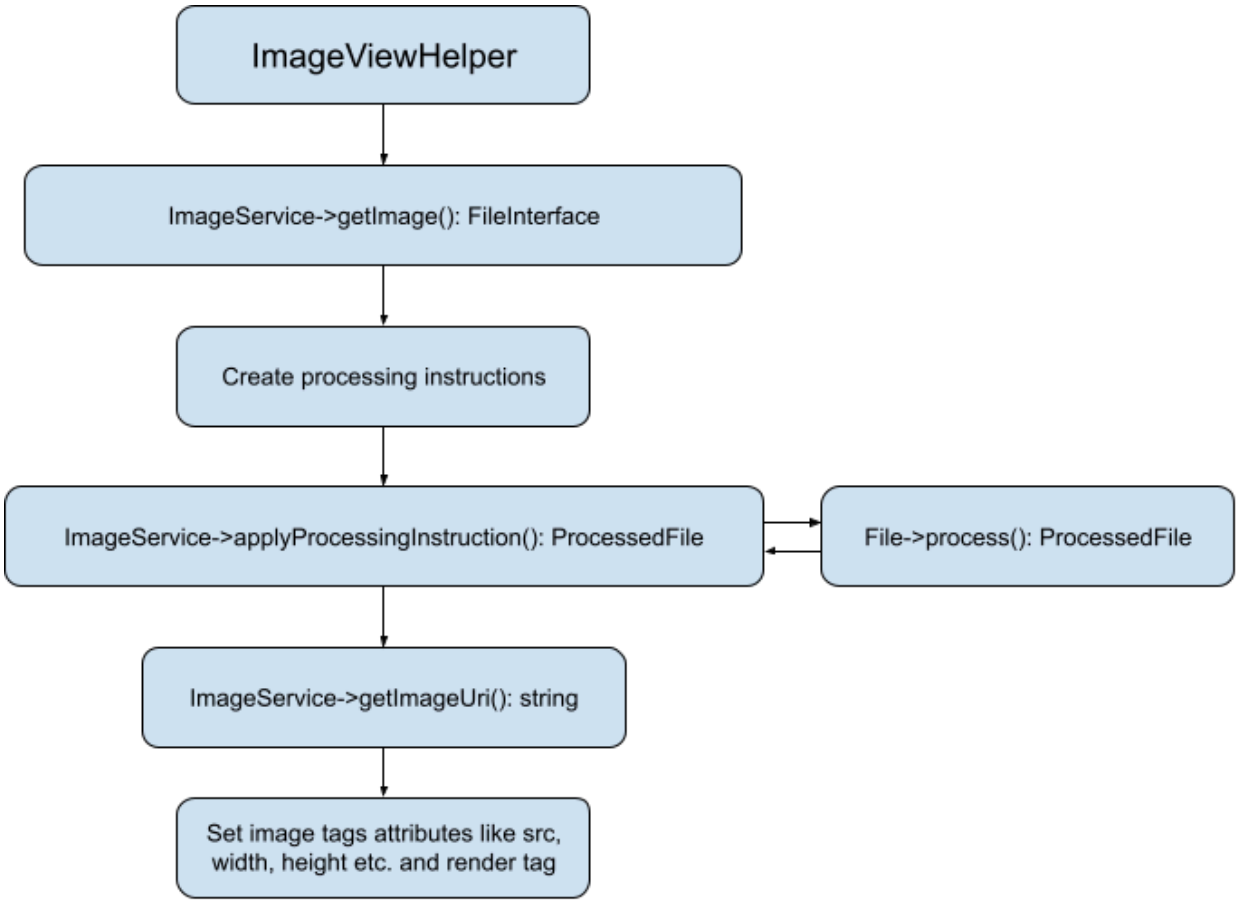


IMAGE and IMG_RESOURCE TypoScript Objects

Opting to render images through TypoScript configuration involves using either an **`IMAGE`** or **`IMG_RESOURCE`** object. These objects, while serving slightly different purposes, adhere to a shared logic for fetching and processing images.

TypoScript's content objects each have a designated class responsible for their rendering, with the classes for these specific objects being:

- **`IMAGE - TYPO3\CMS\Frontend\ContentObject\ImageContentObject`**
- **`IMG_RESOURCE - TYPO3\CMS\Frontend\ContentObject\ImageResourceContentObject`**

The **`IMAGE`** object generates an image tag for a specified file, which can undergo additional processing through various configurations. Files can be specified in multiple ways, such as through a file UID, file reference UID, or a direct path within the file storage system.

This object uses the following template when creating the IMG tag:

```
Python
<img src="###SRC###" width="###WIDTH###" height="###HEIGHT###" ###PARAMS###
###ALTPARAMS### ###SELFCLOSINGTAGSLASH###>
```

When the whole IMG tag is not needed but only the file path of the image, irrespective of whether it has been resized, the **`IMG_RESOURCE`** content object could be used. This example shows how to get and print on the page relative path to image file with UID 3, which is in the /fileadmin storage inside the /test directory:

```
Python
page.30 = IMG_RESOURCE
page.30 {
 file = 3
}

Output:

/fileadmin/test/test-image.png
```

Since the file is an image resource (**`\TYPO3\CMS\Core\Imaging\ImageResource`**), it offers several properties that can be modified to process the image in various ways, such as rotating, setting quality, cropping, and more. This example shows how to process the file to ensure it has a *width* (cropped) of 150 pixels, a *quality* set to 90, and the extension *jpg*:

```
Python
page.30 = IMG_RESOURCE
page.30 {
 file = 3
 file.width = 150c
 file.ext = jpg
 file.params = -quality 90
}

Output:

/fileadmin/_processed_/c/3/csm_test-image_5a629681e5.jpg
```

Documentation:

https://docs.typo3.org/m/typo3/reference-typoscript/main/en-us/ContentObjects/Image/Index.html

https://docs.typo3.org/m/typo3/reference-typoscript/main/en-us/ContentObjects/ImgResource/Index.html
https://docs.typo3.org/m/typo3/reference-typoscript/main/en-us/Functions/Imgresource.html

FAL

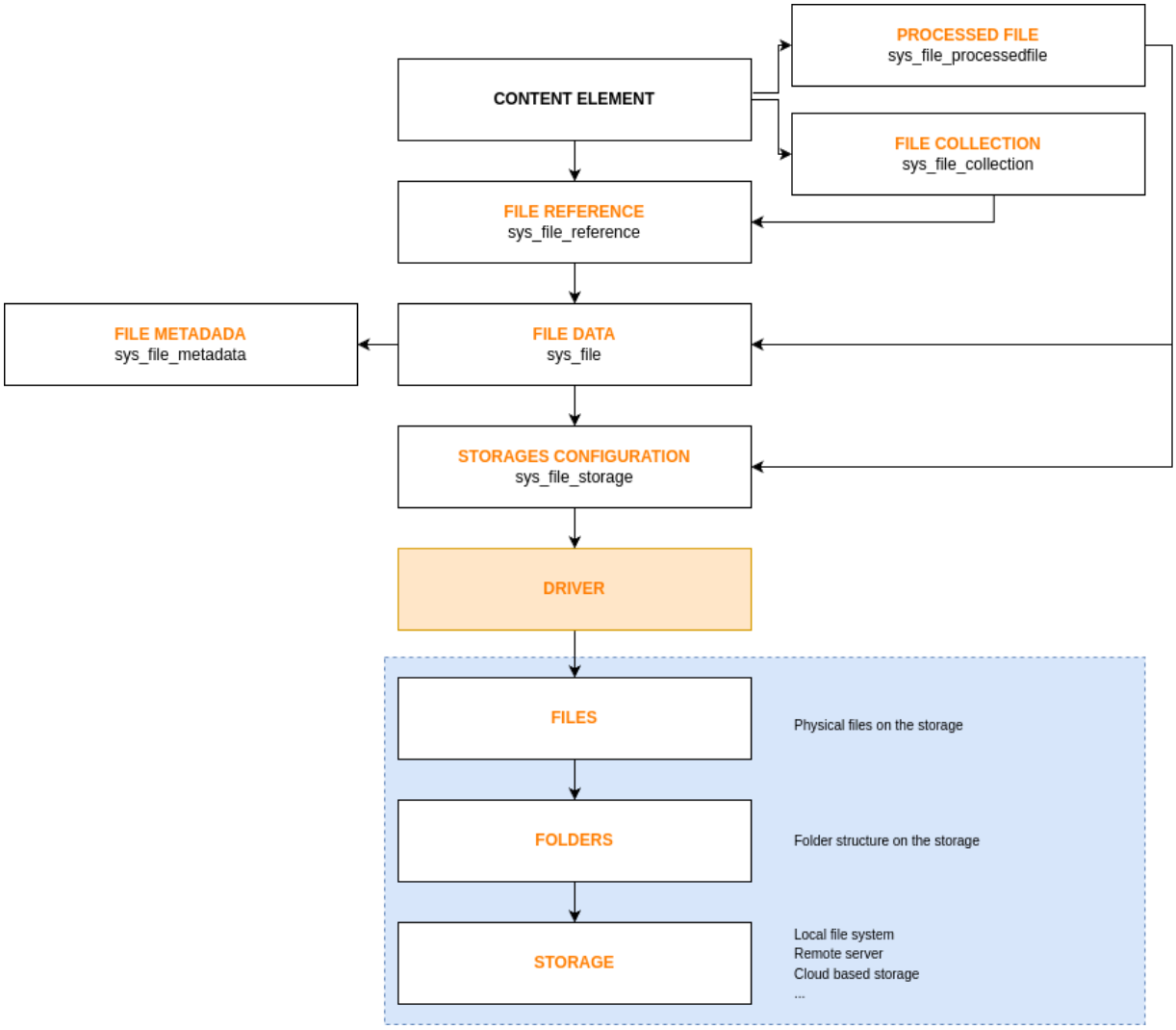


Image: General Structure of FAL Architecture

Processing Files (Images) Across Different Systems

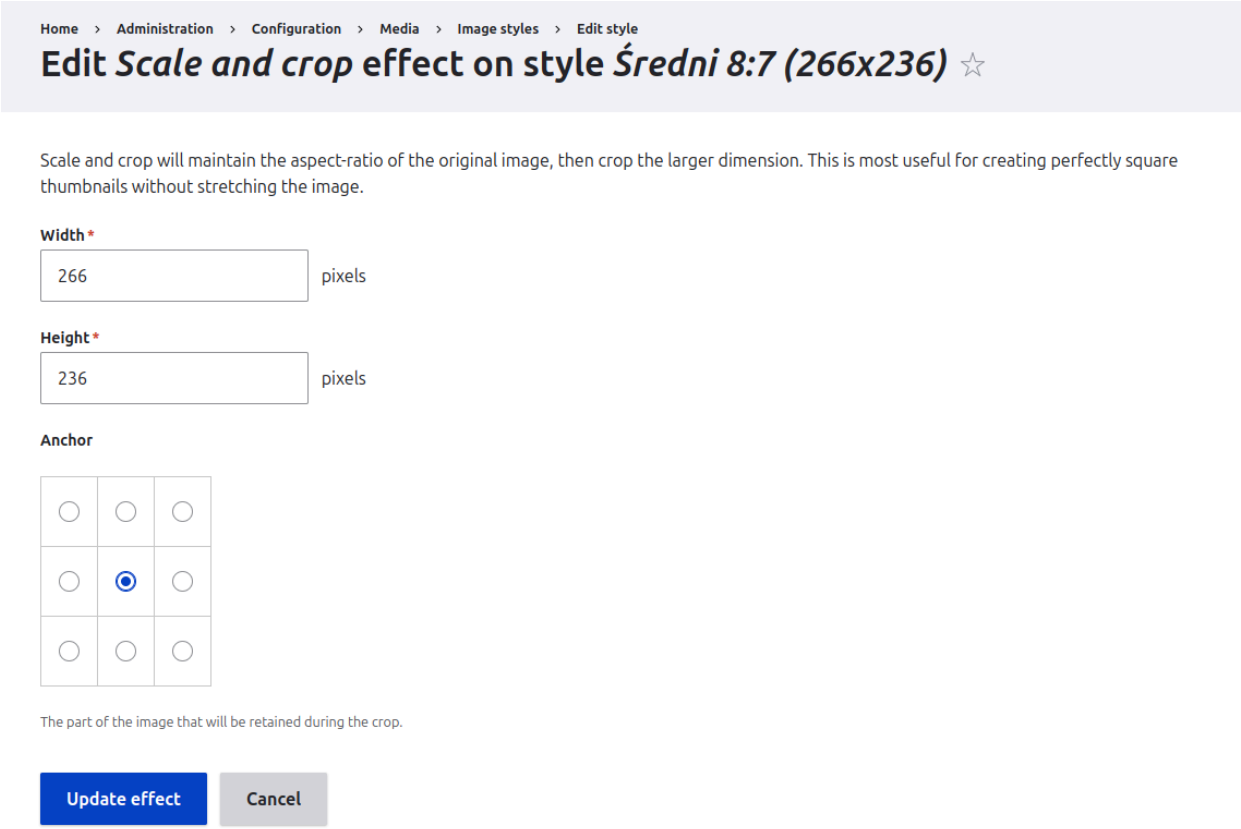
Drupal

By default public files are stored in the local file system for the default site like *sites/default/files*:

```
None
/sites/default/files/styles/medium_8_7/public/chocolate-brownie-umami.jpg?itok=BwdGtF_r
```

The file path includes the style (preset) used like *medium_8_7*:

Used effect Scale and crop 266x236:



If configured the whole *srcset* can be rendered for an image:

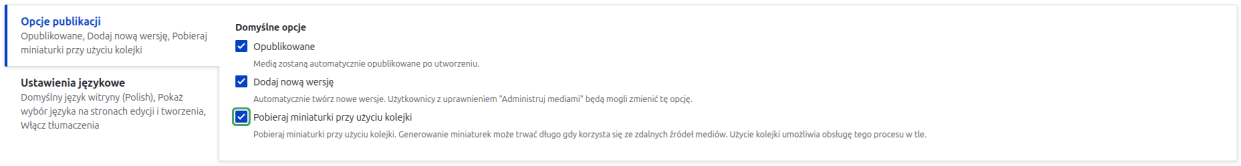
None

```
<img loading="lazy"
srcset="/sites/default/files/styles/medium_3_2_600x400/public/oatmeal-fruit-syrup-topping.jpg?itok=TglmojLy 600w,
/sites/default/files/styles/large_3_2_768x512/public/oatmeal-fruit-syrup-topping.jpg?itok=AIjyqSUB 768w,
/sites/default/files/styles/medium_3_2_2x/public/oatmeal-fruit-syrup-topping.jpg?itok=sqvF2wHw 1200w,
/sites/default/files/styles/large_3_2_2x/public/oatmeal-fruit-syrup-topping.jpg?itok=V6s1VyXJ 1536w" sizes="100vw" width="600" height="400"
src="/sites/default/files/styles/large_3_2_768x512/public/oatmeal-fruit-syrup-topping.jpg?itok=AIjyqSUB" alt="Oatmeal topped with a vibrant mix of berries, nuts, and seeds">
```

The default options like the *path*, *private* or *temporary* files path can be set through the *settings.php*, which is stored within the site configuration directory like: *sites/default/settings.php*.

Queued thumbnail rendering

The given media type offers an option to render thumbnails using the queue. This is useful when the file file is stored on the remote server and downloading the thumbnail or file metadata might be slow and the local data must be saved (using db transaction).



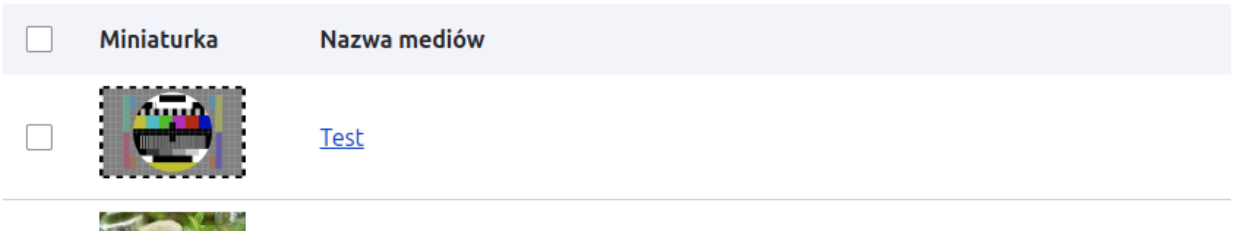
The MediaType entity allows to set the queue_thumbnail_downloads property through setQueueThumbnailDownloadsStatus() function and check it by thumbnailDownloadsAreQueued() function.

The configuration is persisted through MediaTypeForm::submitForm()

When this queue is enabled for the given media type (in our example for images), after the image was uploaded we see the default thumbnail for the file:

```
None
/sites/default/files/styles/thumbnail/public/media-icons/generic/no-thumbnail.png?itok=g1vUw9b6
```

Then if we run the cron, it will check the queue and render proper thumbnail for a file:



The thumbnail download (triggered by cron) is performed by the Drupal\media\Plugin\QueueWorker\ThumbnailDownloader class (*process a queue of media items to fetch their thumbnails*).

More about file structure on the server and files handling: https://www.drupal.org/docs/8/core/modules/file/overview#s-managing-file-locations-and-access

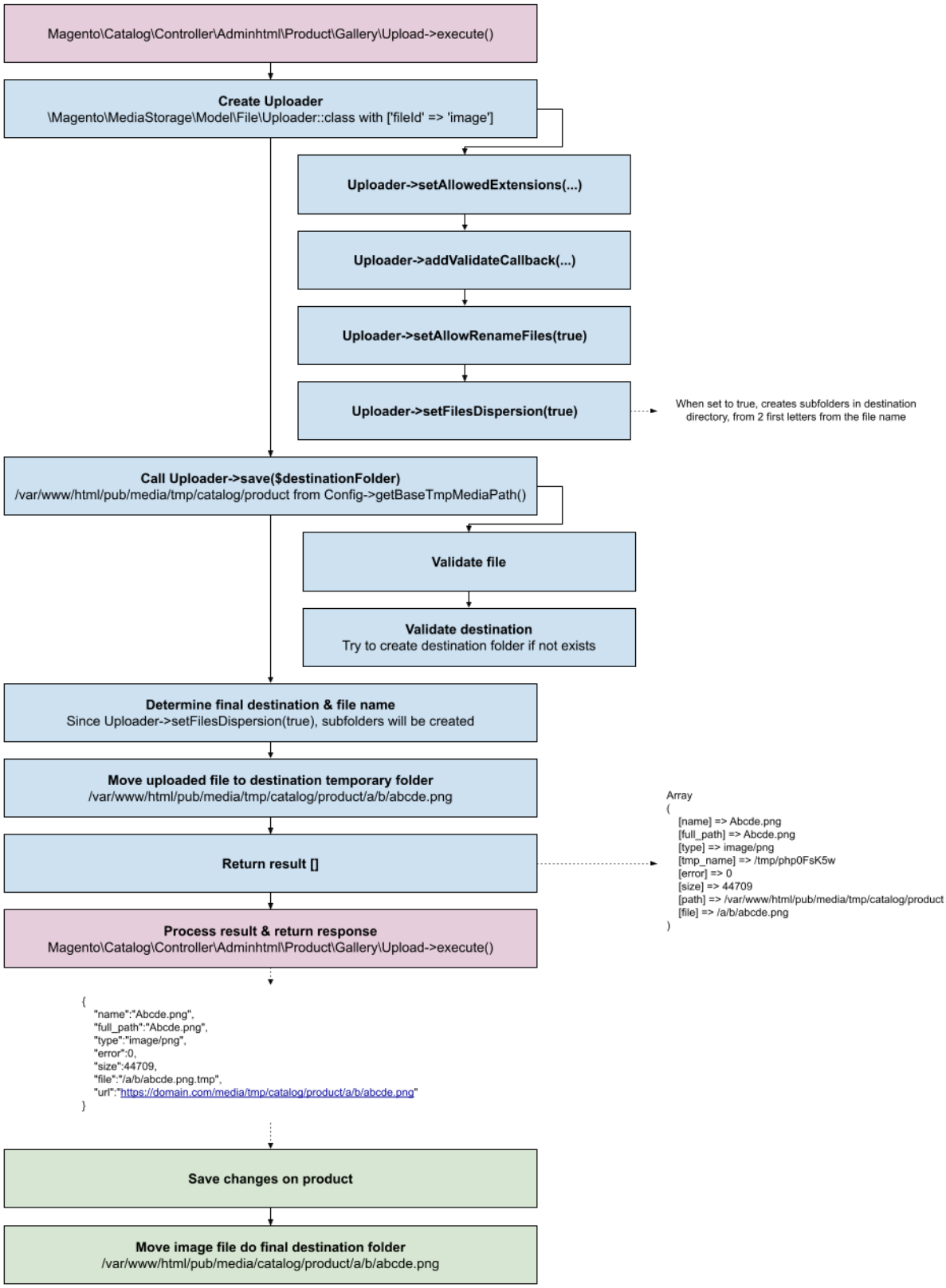
There was discussion regarding some changes but marked as **Won't Fix**: Whether queued or not, update the media thumbnail and metadata before beginning the entity save database transaction

For several years now, there has been an ongoing discussion about changes in how one can fetch thumbnails and associated metadata for a given media type from external storage to make it efficient and optimal: Expose triggering update of media metadata + thumbnail to end users. However, this has not yet been implemented and is currently marked for further development in version 11.x-dev

Magento 2

The Magento 2 is much different when it comes to images handling and processing them (in context of products) than TYPO3 solutions. The main difference is that it does not offer the option to manipulate images (resizing, cropping etc.), but uses the uploaded files directly.

File upload process for product



Summary

- Upload file
- Move uploaded file to temporary directory
- Save changes on product page
- Move file to final directory

Category images

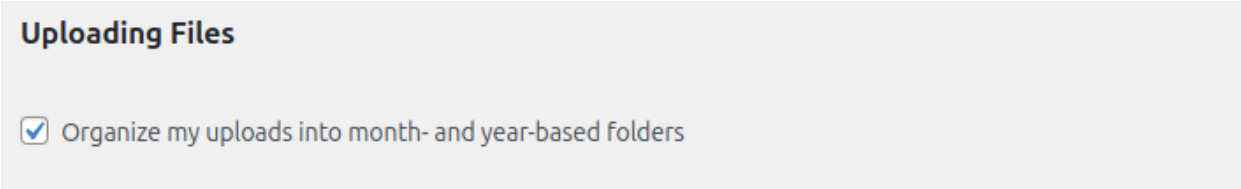
Category image go through similar process, though at first image lands in a temporary folder https://magento2.ddev.site/media/catalog/tmp/category/test-image.png and than is being moved to final destination folder https://magento2.ddev.site/media/catalog/category/test-image.png (without dispersion). If inside the category description image will be used, it will be uploaded into the /var/www/html/pub/media/wysiwyg/ folder.

File upload process for content pages

Here the process is pretty similar. The files are being uploaded into the /var/www/html/pub/media/wysiwyg/ folder with the difference that they are not subjected to the dispersion process which means that there are no subfolders created based on 2 first letters from the file name. After changes were published for a content site, the image url looks like this: https://magento2.ddev.site/media/wysiwyg/new.png

Wordpress

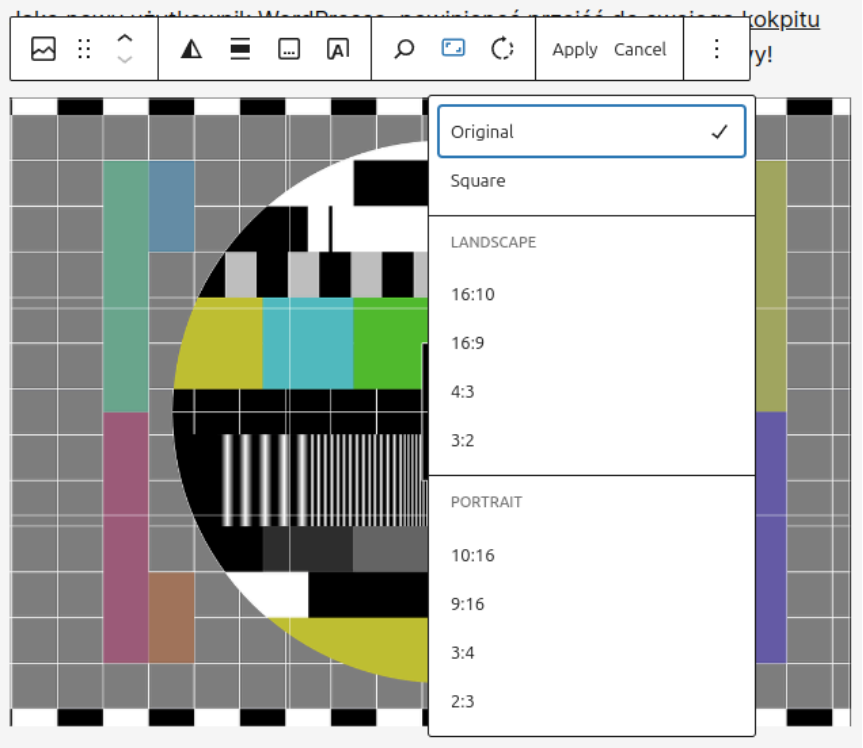
In the context of static pages Wordpress uploads images to the “monthly” folder (under the */wp-content/uploads* path)matching the current month. The files organization can be disabled in Media settings:



The original file, which is rendered in the backend has the following URL:

```
None
https://wordpress.ddev.site/wp-content/uploads/2024/02/test-image-edited.png
```

We can apply custom dimensions for such image, by selecting one of the predefined crop option:



As a result we will have an img tag with srcset set for multiple variants:

```
None
<img fetchpriority="high" decoding="async" width="1067" height="600"
src="https://wordpress.ddev.site/wp-content/uploads/2024/02/test-image-edited-1.png" alt="" class="wp-image-11"
srcset="https://wordpress.ddev.site/wp-content/uploads/2024/02/test-image-edited-1.png 1067w,
https://wordpress.ddev.site/wp-content/uploads/2024/02/test-image-edited-1-300x169.png 300w,
https://wordpress.ddev.site/wp-content/uploads/2024/02/test-image-edited-1-1024x576.png 1024w,
https://wordpress.ddev.site/wp-content/uploads/2024/02/test-image-edited-1-768x432.png 768w" sizes="(max-width: 1067px) 100vw, 1067px">
```

Contentful

General informations regard the images:

- Size of an image uploaded must not exceed 20MB
- Images exceeding the size limit are treated as *assets* and the manipulation features offered by the API are not applicable
- For image assets, the fields.file.url field will point to images.ctfassets.net.

- For other file types, it will point to assets.ctfassets.net.
- For large image assets (greater than 20MB and other asset files greater than 10MB) use the downloads.ctfassets.net.

Typically images are retrieved from *the context of one or more entries* (1-10 levels deep of linked entries, default: 1 if not *include* parameter provided), or by *assets* directly.

Together with an entry data there is only basic information about linked assets and the full asset data can be fetched from the includes field, for example:

```
None
"includes": {
 "Asset": [
  {
   "sys": {
    "type": "Asset",
    "id": "23qqdlTciMGm6IYy224euu",
    ...
   },
   "fields": {
    "title": "The SpaceBurger Photo",
    "file": {
     "fileName": "spaceburger.jpg",
     "contentType": "image/jpeg",
     "details": {
      "image": {
       "width": 550,
       "height": 421
      },
      "size": 205683
     },
     "url":
"//images.ctfassets.net/oc3u7dt7mty5/23qqdlTciMGm6IYy224euu/85514b430c28045a3b2930
ebe15dfcce/spaceburger.jpg"
    }
   }
  },
  ...
 ]
}
```

More:
https://www.contentful.com/developers/docs/references/content-delivery-api/#/reference/assets

When using the Image API it is possible to process the image as desired, referencing the original file through *asset.fields.file.url* and appending desired query parameters like w=200, h=100, fit=thumb,f=top_left, r=180 etc.

Adding image is possible through web app or using API and it involves this 3 steps:

- Creating an asset, which is an entry for the media file.
- Processing an asset, Contentful downloads the media file from the URL supplied and processes it.
- Publishing an asset, making the entry and media file publicly available.

More: https://www.contentful.com/developers/docs/concepts/images/