Architecture

Here you can see a high-level overview of the whole architecture:

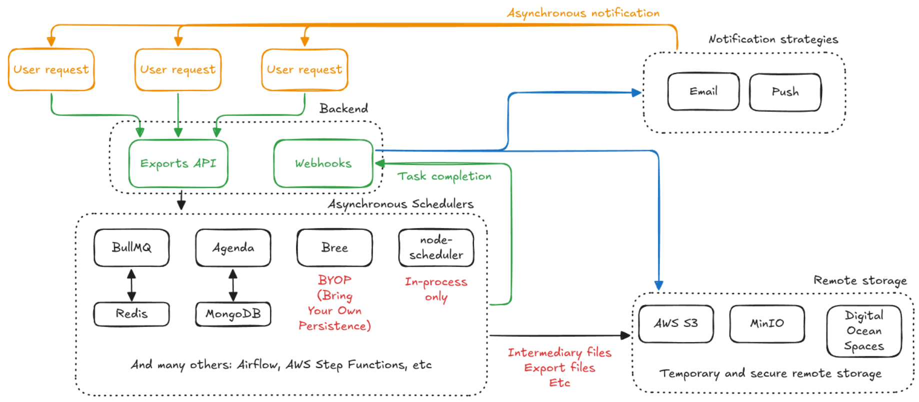

Figure 1: Task 2 Architecture

Exports API (backend)

This might be a monolith or microservices. Either implementation works so the choice is left to requirements and constraints. When the user requests an export, it'll go to the **Exports API** (which might just be a separate route in our existing monolith backend).

The **Exports API** will take in that request, run whatever validations necessary, and then send it over to the **Asynchronous Scheduler**.

Asynchronous Schedulers

Here I've decided to show multiple different options, since they all can achieve the same thing. Here we'd research the different options and based on our current and future requirements, settle on one option. Ideally whichever tool is picked allows for monitoring as well as retries out of the box.

This layer will take care of actually running the long-lived job. Likewise, this layer can also be used to set up periodic long-running tasks that might not be directly triggered by the users.

Scaling of this layer highly depends on the specific tool and storage chosen, but in most tools, it involves a combination of increasing the worker pool to run more concurrent jobs, as well as scaling the storage layer to support more concurrent accesses.

Remote storage

Likewise, here I've decided to show multiple options, but you'd also settle on one solution.

The reason why you need a remote storage layer is that it's often very difficult (or even impossible) to serve large/asynchronously-created files directly from the backend server. Even if those servers/containers had infinite, non-volatile storage, that might leave you open to different kind of attacks.

For this reason, it's often a good idea to use a separate and non-volatile storage layer where you can control both the longevity of the files, as well as access to those files. AWS S3 offers this with presigned URLs, that allow temporary access to download/upload files to a given S3 bucket, without exposing any other files in that bucket, and also auditing all accesses to that bucket/file.

This layer can also be used for temporary intermediary files that the **Asynchronous Schedulers** layer might need to complete a job.

Webhooks (backend)

Another part of the backend, might just be a different route in our backend monolith, or a whole new microservice. The **Webhooks** would get notified by our **Asynchronous Scheduler** job that a given job is finished, along with any necessary metadata. This API could also be used to report job failures.

Once a job has been completed (success or failure), this part would also be in charge of deciding the notification strategy, and then sending it out to the user. The notification strategies might just be set through the program configuration, or might be set by the user picking whether they want email, app notifications, in web notifications, etc.

For this, you'd likely want to have different notification templates, email, push, etc, then pick the notification service (if any and necessary), create a temporary and secure access to the given export file in the **Remote storage**, then notify the user.

Notification strategies

Here I've decided to model it as a separate part, as many notification strategies, like emails or mobile push notifications, will often require cumbersome third-party services. For something like email, services like AWS SES, Mailgun, Sendgrid and others come to mind (setting up your own email server is also possible, but quite an undertaking due to reputation checks).

For mobile push notifications, services like AWS SNS also offer integrations.