REQUEST FOR COMMENTS

Product and Theme Data Management

Problem & Context

Throughout the Shop and Editor ecosystem, the “source of truth” for our product and creation data resides in a number of GIT Repositories which contain manually maintained, hardcoded JSON files.

Two of the main ones being the Rollercoaster Data Modelling repository, which contains all product specific data from physical definitions to marketing and sales definitions. As well as the Design Data repository, which contains assets available to customers in the Editor, such as Backgrounds, Decorations and Fonts.

https://github.com/photobox/rollercoaster-data-modelling

https://github.com/photobox/ecom-design-data

GIT was chosen originally for a few reasons, namely it allowed the development of the customer facing platform to progress at a faster pace. It was also chosen due to the inherent versioning capabilities of GIT along with the familiarity that most developers have with it.

Given that all of the product and design data is stored inside GIT repositories, only developers have access. This method of data persistence has created a number of issues. It is also extremely inefficient and has dramatically slowed down both product launches and developer capacity, of which could otherwise be used to deliver more customer value.

We need to remove this bottleneck and remove the need for dedicated engineering resources. As such we have concluded that we need to build a UI that can be used by the wider business to perform product and design data updates without input or assistance from engineering. The solution will require an API and Database as a persistent store for the data.

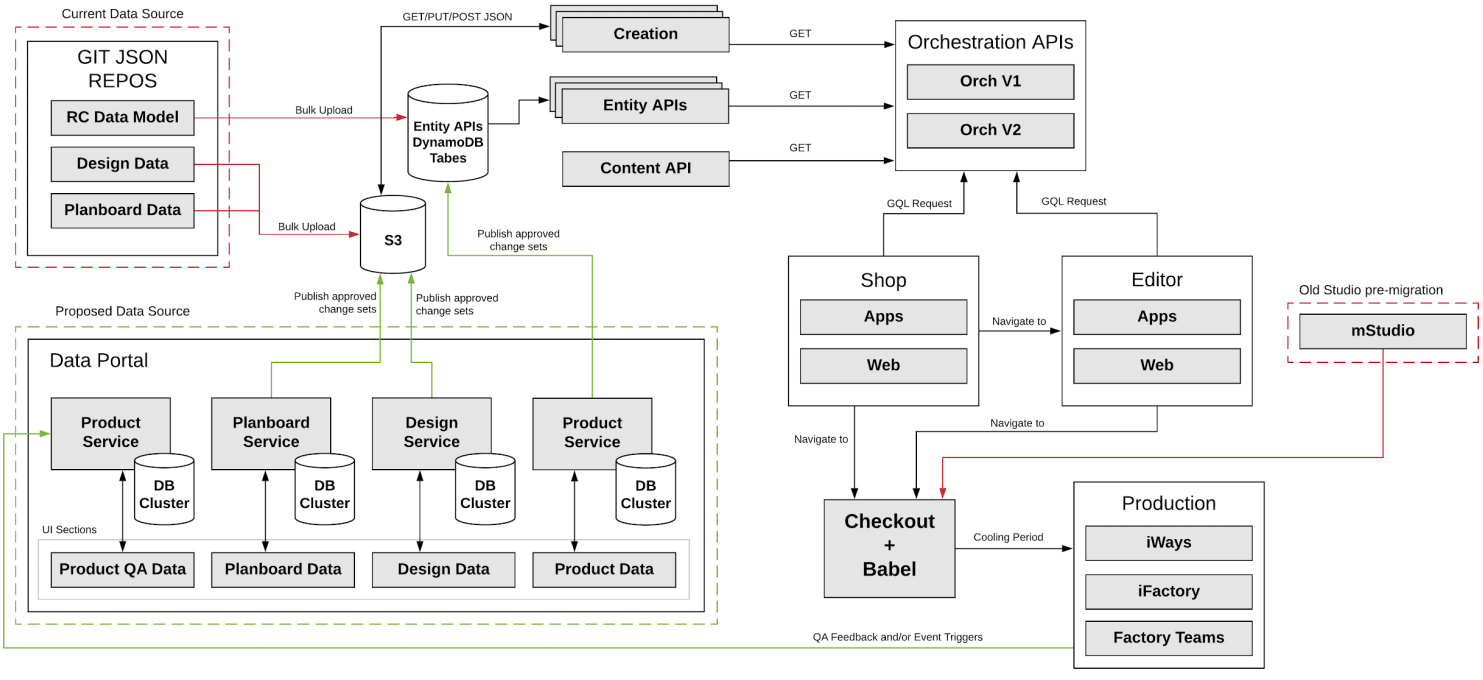
The main purpose of this RFC is to outline how the data, once migrated, will be managed. There are a number of points to take into consideration when thinking about how the data will flow from source to destination. We also need to think about the different types of users that will come to manage it in both short and longer term scenarios. The document will cover the following points and additional follow-up RFCs may be required off the back of any section.

1. Database versioning and backups
2. Saved vs published data
3. Keeping different environments in Sync
4. Bulk updates
5. Transitioning to the new model
6. Integrations and potential overlap with CommerceTools

Solution

Architecture

The is the top-level architecture diagram of the ecom ecosystem. The red sections are to be replaced by the green sections as part of the work scoped out at the beginning of the mStudio migration project (planboard data changes likely to be descoped in the short term). It has been included to highlight where the in the ecosystem data flow from and to.



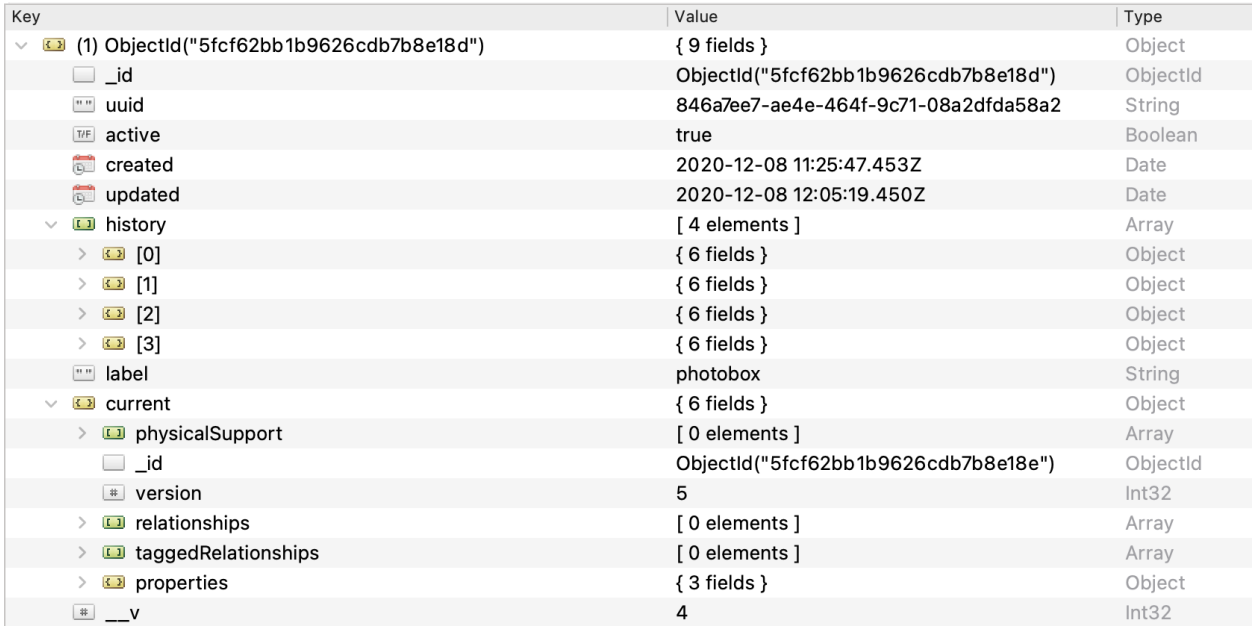
Database version and backups

A previous RFC for Themes outlined the high level architecture for managing the design data in a UI. The product data will follow a very similar pattern for consistency. This previous RFC was reviewed by the DBOps team and, given the nature of our product and design data, it was agreed that DocumentDB would be the most appropriate solution for the Database.

DocumentDB is an Amazon managed NO-SQL Document database that is forked from the widely adopted MongoDB. It is a document-oriented database that provides native indexes, elastic scaling along with a powerful query language. Being a document-oriented database it is ideally suited to JSON like documents such as the ones we currently maintain in files. The JSON-like structure afforded by Mongo does present a rather elegant way in which we can handle document versioning.

Documents can have a self-contained revision history. There are a couple of ways you can store historical versions:

1. Each object in the history would be the complete document from that point in time.
2. Each object in the history would just be the change set applied.



Key	Value	Type
(1) ObjectId("5fcf62bb1b9626cdb7b8e18d")	{ 9 fields }	Object
_id	ObjectId("5fcf62bb1b9626cdb7b8e18d")	ObjectId
uuid	846a7ee7-ae4e-464f-9c71-08a2dfda58a2	String
active	true	Boolean
created	2020-12-08 11:25:47.453Z	Date
updated	2020-12-08 12:05:19.450Z	Date
history	[4 elements]	Array
[0]	{ 6 fields }	Object
[1]	{ 6 fields }	Object
[2]	{ 6 fields }	Object
[3]	{ 6 fields }	Object
label	photobox	String
current	{ 6 fields }	Object
physicalSupport	[0 elements]	Array
_id	ObjectId("5fcf62bb1b9626cdb7b8e18e")	ObjectId
version	5	Int32
relationships	[0 elements]	Array
taggedRelationships	[0 elements]	Array
properties	{ 3 fields }	Object
__v	4	Int32

As with anything there are pros and cons to the two approaches. The pro to only storing the changeset is that you're reducing how much data you're duplicating in the history. Documents have a limit of 16MB so this does potentially become important. The downside is complexity, to revert to a previous version you need to apply all the change sets in order. For this reason the recommendation would be to go with the first option and store the entire document. It is the simpler implementation and as our individual documents are relatively small in size the limit on storage should be minimal. To get around this further we can also look at setting a limit on the number of historical documents that can be stored. Having time/quantity based limits on version history is a common pattern.

When querying the API we would always return the current document by default.

Alternative Approach

Documents are stored in Collections (similar to Tables in a traditional SQL DB), another solution could be to have a "shadow collection" for each. The main collection would contain the most recent data for the stored document. The "shadow collection" would contain every historical record. This allows us to separate the current and historical data and removes the concern around max document size. However, it is a lot more overhead to maintain.

Database backup strategies are discussed later in this document under the Redundancy section.

Saved vs published data

In the current ecosystem data is managed by engineers and, through Jenkins pipelines or deployment scripts, flows from development to staging and once tested and approved, production. However, eventually the target audience for the new platform will not be engineers but any internal Photobox Group employees with the proper permissions to log into the new data portal UI and manage the data.

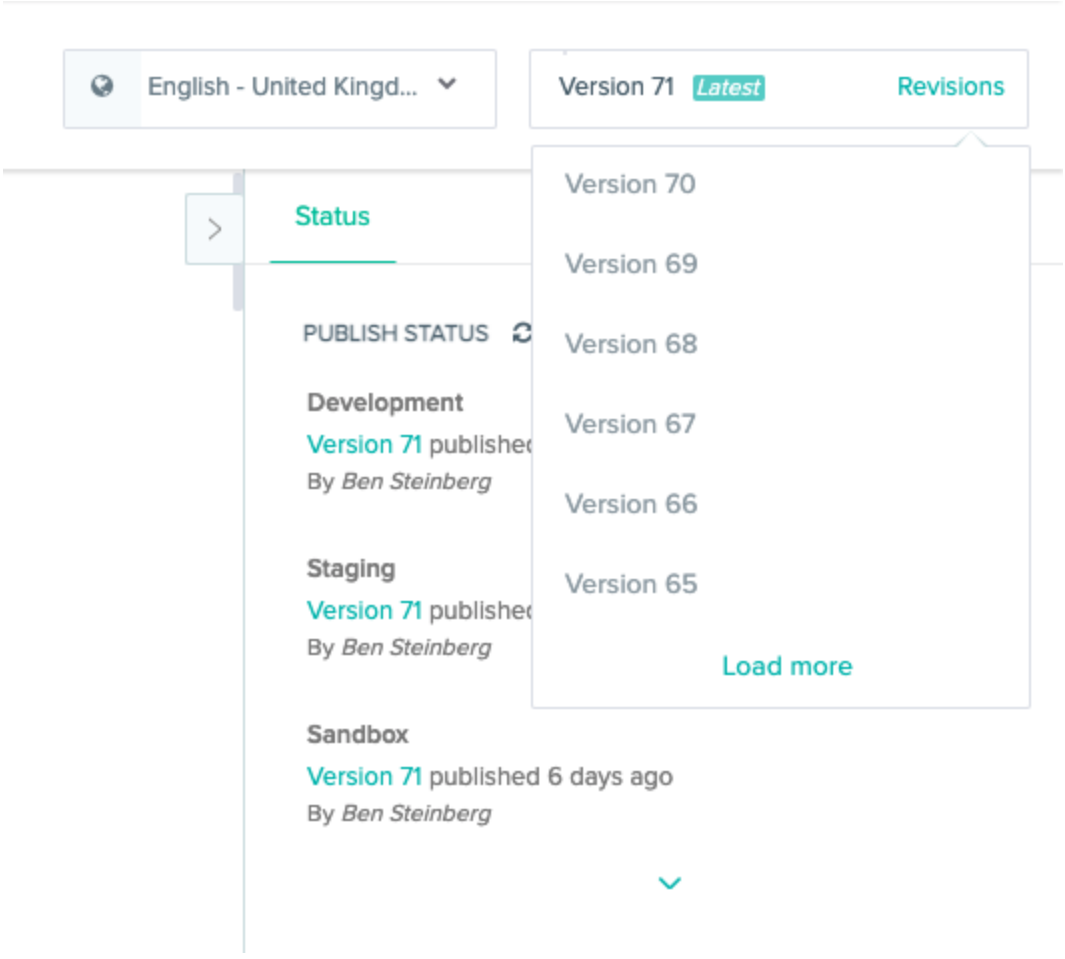
The other issue we have at the moment is that product data testing, particularly physical print testing, can only happen in production. We could potentially look at implementing a flow to get Staging working but physical print QA can take a long

time and having data which is still in QA will ultimately get wiped out on staging through other BAU updates and we would need to constantly keep it in sync.

The proposed solution to the above is for the users of the new data portal to work on data changes in the production environment. Of course, we need to make sure that we can preview and approve any changes we make because they're made like to customers. This means we need a concept of a "saved" stated and "published" state in the data.

The solution here follows on very nicely from the previous discussions around keeping historical versions of documents in the database. We can surface these versions in the UI (a model similar to how ContentStack manages versions for published data) to make it very obvious to the users which version is currently published. Saved but not yet published data will be previewable on the site via query string parameters to pull through versioned data rather than the live published version of the data.

There shouldn't be a need to publish across environments but this will be discussed in a bit more detail further down.



With the above approach the flow for both Product and Design Data would be:

1. A user makes an update to a document and clicks save. A new version of the document is created.
2. The new version is exported out to the relative services (e.g. to S3 and the design service for design data and the entity apis for the product data).
3. The new version can be previewed on the relevant production domain by passing a query parameter to request that new version specifically. (The data portal could even have a preview link out for ease).
4. Once approved the user can publish that version to all / relevant locales if applicable.
5. Once published, the published version is updated in the database. The timestamp and author of the published action is also stored. The relevant updates are made in the client services to point the live data to the new version of the data (e.g. for the design data server the new s3 file will be copied to the current file).

In future iterations of the data portal we could even look to have scheduled publish actions.

Keeping different environments in Sync

Whilst having users of the data portal only working in the production environment, with saved and published data is a very nice flow for data management and QA, it does present an issue for developers.

There will be times where new features are being worked on in development or staging and data, which was only added in production, is needed. Or perhaps there is an issue that needs to be debugged out of production. There are a few options here:

1. Similar to ContentStack, we allow data to be published across environments. The issue with this is that it relies on the users remembering to select all the environments. Alternatively, as we're only talking about going from production to dev/staging, we could automatically publish any updates to those environments but it will slow down the process and be more costly.

2. We can look into DocumentDB Change Streams, change streams allow you to watch for data changes or any kind and then perform some action or process on that data. The issue here is cost, we'd need to have an EC2 (or similar stood up server) which would watch all the data constantly, it maybe possible that something could go wrong or be missed and the data would get out of sync anyway.

3. We have a scheduled runner, either nightly or more often, which would sync the production databases to dev and staging (potentially we might even be able to copy the db snapshots from production and restore those in dev/staging). The downside here is that the dev and staging databases would only periodically get updated, but the approach is a lot simpler to implement and maintain.

As a starting point my recommendation is always to keep things as cheap and simple as possible and then iterate. As such option 3 would be the recommendation here. We could start with nightly data syncs and see how this worked as a workflow. If engineers found it a big pain point we could either increase the rate of data sync processes, allow them to be triggered manually or look at an entirely different solution.

Bulk updates

Moving away from a process of managing data in JSON files to instead using a UI has a number of benefits. However, it does potentially have one downside that when bulk updates are needed, it is not as simple as a find / replace in a JSON file. That said once a find/replace might be easier it is far more prone to human error, hence the decision to move away.

With the new system we want to make sure bulk updates are still possible and whilst the process might be a bit more effort, it shouldn't impact productivity for the engineer. Time added setting the data should be saved in time validating the changes and deploying time.

There are a couple of ways we can achieve. First, is by designing a UI that allows some data to be set in bulk. E.g. have an area in the portal where you can set a stock level and apply it to multiple products/locales (instead of setting the stock solely on a product by product basis).

The API will need to be built to accept bulk updates to support this UI. We will build this “api first”, the API should have the ability to bulk update all entity types, regardless of whether a related section of the UI exists yet. The benefit here is that if there is a bulk update which the UI does not yet support, short term scripts can be written to call the API endpoints directly with bulk updates. Going via the API rather than direct to the DB means we can ensure all data updates are validated accurately.

Transitioning to the new model

Another consideration is how to manage on-going work in the existing data repositories whilst the new systems are being built and worked on and when exactly there should be a “hard switch” for all teams to the new system.

We have built some idempotent import scripts which can be run to re-import the latest data into the database. However, the issue is at the point we start adding new data to the new system, merging the updates from to sources becomes very tricky.

Design data

With the design data the updates are less regular and managed by fewer people and teams. It will be less complicated to agree to a switch point and the recommendation is to do a hard switch at the earliest convenience. We need to be sure the switch won’t impact the teams needing to use it, even if the data portal is not “complete”, we should be able to make BAU updates as normal.

The data portal will be used to create themes, both “default” themes as part of the design toolkit for products as well as themes that will be browsed via the Shop. However, we can do a switch before the functionality exists simply by ensuring we can export the “design design” which is the JSON that is currently being used for all products and creations. The plan is to initially just have this exporting, switch away from the JSON repository and then move forward with Themes.

Product data

There is some potential overlap of requirements here with the data being proposed to move to CommerceTools. These dependencies need to be aligned and agreed before we can move forward with a decision on what product data the portal needs to be managing and when the switch from the existing JSON model should happen. These dependencies and integrations are discussed more in the next section.

Dependencies & Integration

Integrations and potential overlap with CommerceTools

At the moment there is still a lot of discovery and POC work happening with CommerceTools. So far the following has been highlighted by the Core team as entities of the product data that they see being moved over to CommerceTools:

- Product
- Variant
- Range
- Stock
- LODs
- Recommendation
- Options
- Option Groups

In terms of what would remain to move into the data portal it would be mostly physical product definitions, namely:

- Physical Variants
- Physical Supports

If this is the case it reduces the scope of work to enable Product Data management in the data portal but it does raise some additional questions around timelines of objectives.

Initially the idea to move Product Data came from the long term goal of being able to add new products via a UI. For the time being should all product data (except the physical entities mentioned) remain where it is until the CommerceTools integration is up and running?

Design updates for Cards and the Editor Core team

The Editor Core team are looking at themes for cards, we have initiated a number of discussions and meetings around how we can best align our goals, timelines and backlog to support each other.

Initially the data portal will be built very much as an MVP. It will have the foundation aspects discussed in this document around how data is managed and a basic form-based UI for using / testing the flow. The Editor Core will then work on top of this to build out a nicer more optimised UI for building themes to be used by the wider team.

Infrastructure

The infrastructure has been outlined in a previous RFC.

Now that the Database solution has been agreed we know that this will require three new managed DocumentDB Clusters to be created along with a dedicated VPC and security group. One of each for development, staging and finally production.

Scale & Performance

The data portal is strictly an internal tool and as such there are no real performance concerns on that side. That said, the solution will still use serverless technology so we can be confident it will scale nicely to the limits of Lambda and the throughput limits of DocumentDB.

In future we might want to update orchestration layers to call the data portal API directly to get data and remove the complexity of additional services. If and when this happens, the serverless approach should support such a model but of course some load testing would be needed and likely some work to look at the size and scaling options of the database.

Reliability

Amazon DocumentDB is designed for 99.99% availability and replicates six copies of your data across three AWS Availability Zones (AZs). You can use AWS Database Migration Service (DMS) for free (for six months) to easily migrate your on-premises or Amazon Elastic Compute Cloud (EC2) MongoDB databases to Amazon DocumentDB with virtually no downtime.

Redundancy

DocumentDB continuously backs up to S3 for 1–35 days, allowing us to quickly restore to any point within the backup retention period. DocumentDB also takes automatic snapshots of the data as part of this continuous backup process.

Along with general backups of the database we will also keep track of data changes within the data so we can track what was changed when, by whom and, if needed, revert back to an older version of the data.

Monitoring & Instrumentation

This is largely non-customer facing so we will just use CloudWatch alerts and logs to monitor any issues.

Failure Scenarios

A regular backup policy will be inplace to ensure we have a constant way to rollback to a recent dataset in case of a disaster.

The Data Portal will publish data out to other services in the short term. This gives us a nice safety net because in a situation where some incorrect or broken data is published to production, at the same time the database falls over meaning that we can not correct the issue, we can still manually correct it if we need to.

Security

Access to the Database will be strictly limited to the API. The API itself will be secured heavily with Cognito. Only authenticated users will be able to modify or publish data.

Privacy

No customer data will be stored as such there will be no privacy concerns.

Operational Implications

A lot of the work being done here will eventually be picked up by other teams, i.e. the Editor Team might pick up the theme and design data management side and the Core Team might pick up whatever remains of the product data management.

We're in constant communication with both teams but we need to make sure we remain aligned on what is being built, how it is being built and why.

Risks & Open Questions

Most of the risks and open questions at the moment are around inter-team dependencies. We need to make sure timelines and objectives are understood and aligned.

At the moment a big piece is how Themes will work for, both generally and for Cards with CommerceTools:

1. How will themes be set up in CommerceTools?
2. How will we link a creative theme to a marketable theme?
3. How will we manage product (and more specifically) variant associations to themes?
4. How will we define default product themes?

There is still a lot of uncertainty here but initial conversations have been kicked off and a separate RFC is being worked on.

Some initial ideas and thoughts:

1. In CommerceTools (and/or the existing RCDM) we need a property added to Products and Variants e.g. “defaultThemeId” which will be used to define which default theme should be used in the Editor for that particular product/variant.
2. As a general rule we should attempt to keep Themes at a product level but if necessary we need the ability to assign at a variant level.
3. When the Editor loads a variant (without a preselected theme) it will look for a variant theme id, if it can not find one it should fall back to the product theme id.
4. Many additional themes should be available to search and browse in the Shop, these would be defined as Categories in CommerceTools.
5. When a Theme is created in the data portal, on save(?) we would need to make a number of API calls to CommerceTools to create the theme category and set up the required product/variant associations. A possible workflow might be:

- A new theme is added to the data portal:
    - A user adds a name, description, thumbnail (would these need to go up to ContentStack?)
    - A user adds the product and variant associations by RC IDs
    - On Save (as we want to be able to preview - is this an issue in CommerceTools as you can only save/publish categories?) we make a call to CommerceTools to create new theme category and sub category (include external theme id attribute)
    - Make additional calls to CommerceTools to update all associated products with the new CT theme sub-category
    - For variant associations we need to update a custom attribute on the product with links to the themes that can be picked up by variants?

All of the above is very much still in discovery and is all still open to change.