💡 RFC: Document PDF Rendering Service (Feb. 2020)

Owner

@Shaun H and Documents & Messaging Team

Stakeholders

- Docs & Msg
  - [ ] @Aaron S
- Compliance
  - [ ] @Faiz A
- Security
  - [ ] @Alan F has a customer commit
- Infra/Platform
  - [ ] @Edwin Z
- Eurofleet
  - [ ] @Ewelina S
  - [ ] @Matthew H
- Devops
  - [ ] Jon San Miguel

Background

The Docs&Msg team has a commit for Millennial Transport to provide a PDF export of documents via API. We would like to create a PDF rendering process that can demonstrate a capability that other teams would find repeatable for their own uses.

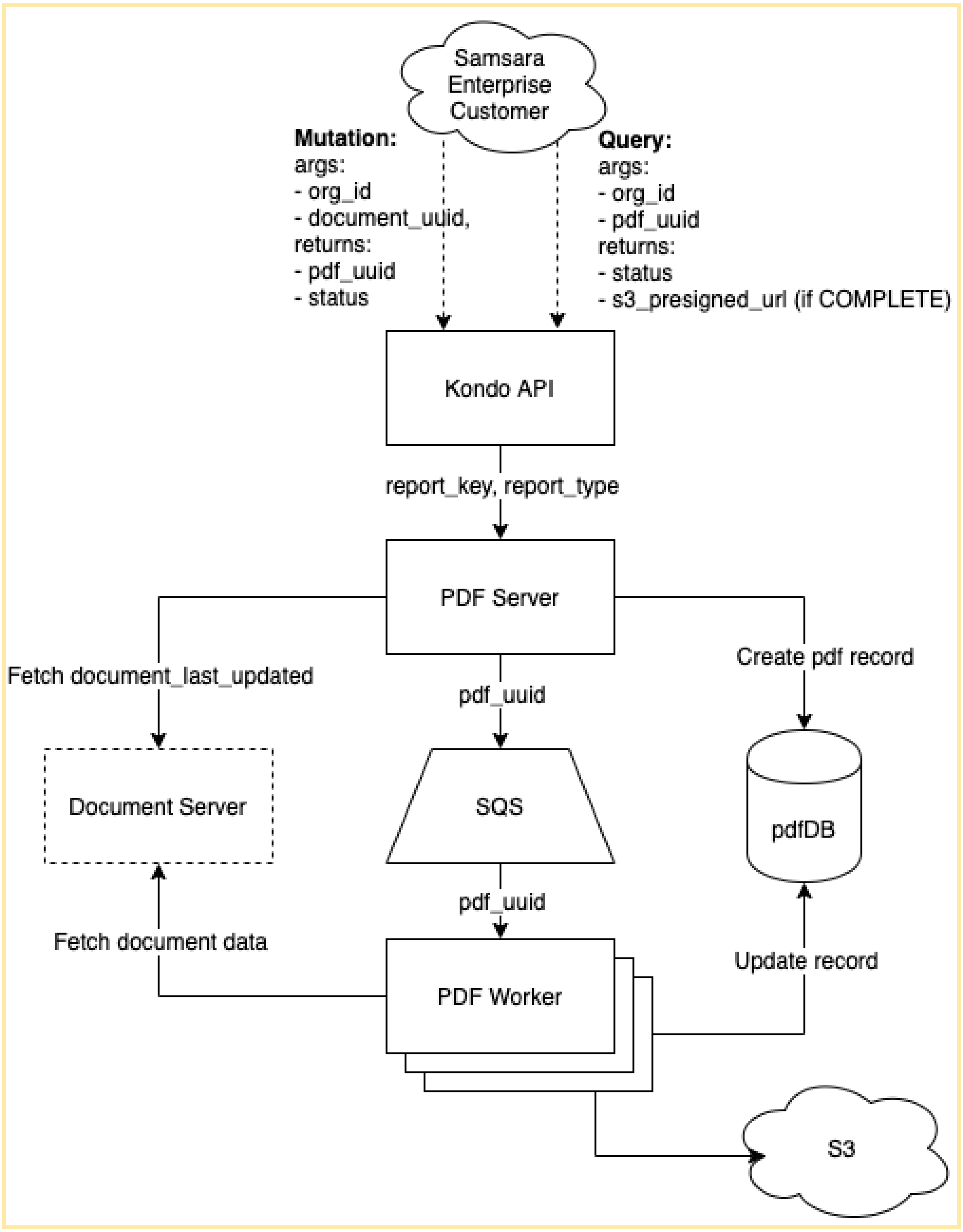
Goals

1. Create a system that renders a Driver Document (including images) to PDF
2. Demonstrate a possible solution for other teams that want to render PDFs
3. Attempt to provide pathway to extend service to other report types

Non-Goals

- Creating a platform for non-technical Samsarians (or customers) to author reports. Each new report template will require an engineer to write code.

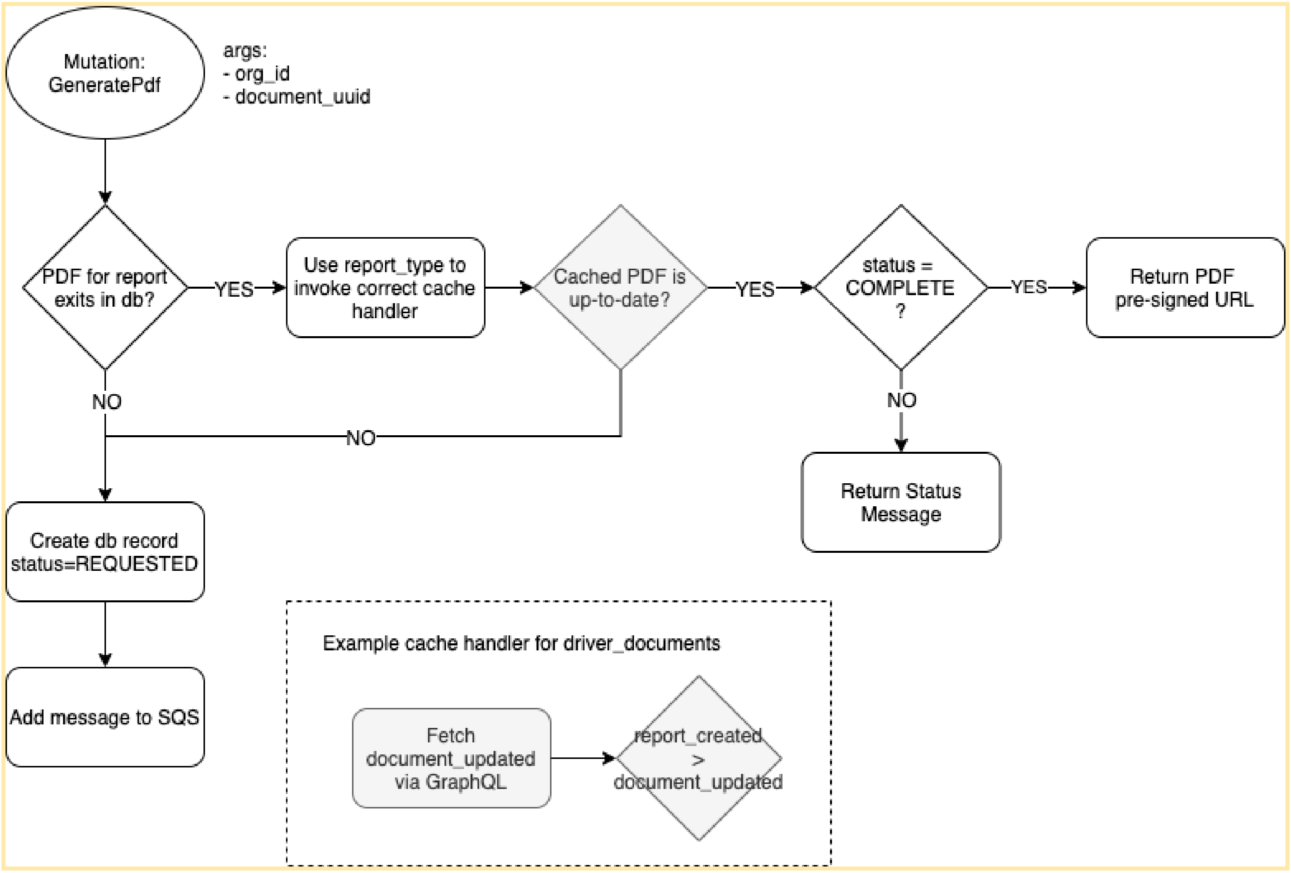
Architecture



Samsara Enterprise Customers - Customers that are already using our v1/kondo apis to programmatically access data. In this design it will be necessary for a customer to poll the S3 pre-signed url periodically until the object is found. In practice I expect PDF creation to be a fast process so the customer should be able to finish polling within a few seconds.

Kondo API - The entry point for the first use-case is our public API layer. This is a crucial component of the design as it builds on our existing infrastructure to handle authentication, data-access, and S3 pre-signed url creation.
+ API Design: Document PDF Generation (April 2020)

Documents Server/PDF Service - gRPC endpoints on driverdocumentsserver exposed via graphQL for creating and querying pdfs. Will create records in the pdf table and handle checking for existing cached PDFs that are up-to-date. Generates pre-signed S3 urls for completed PDFs. Hands pdf generation requests off to SQS to be handled by a worker. Option for client to provide a flag which will force creation of a new PDF regardless if an up-to-date one exists.

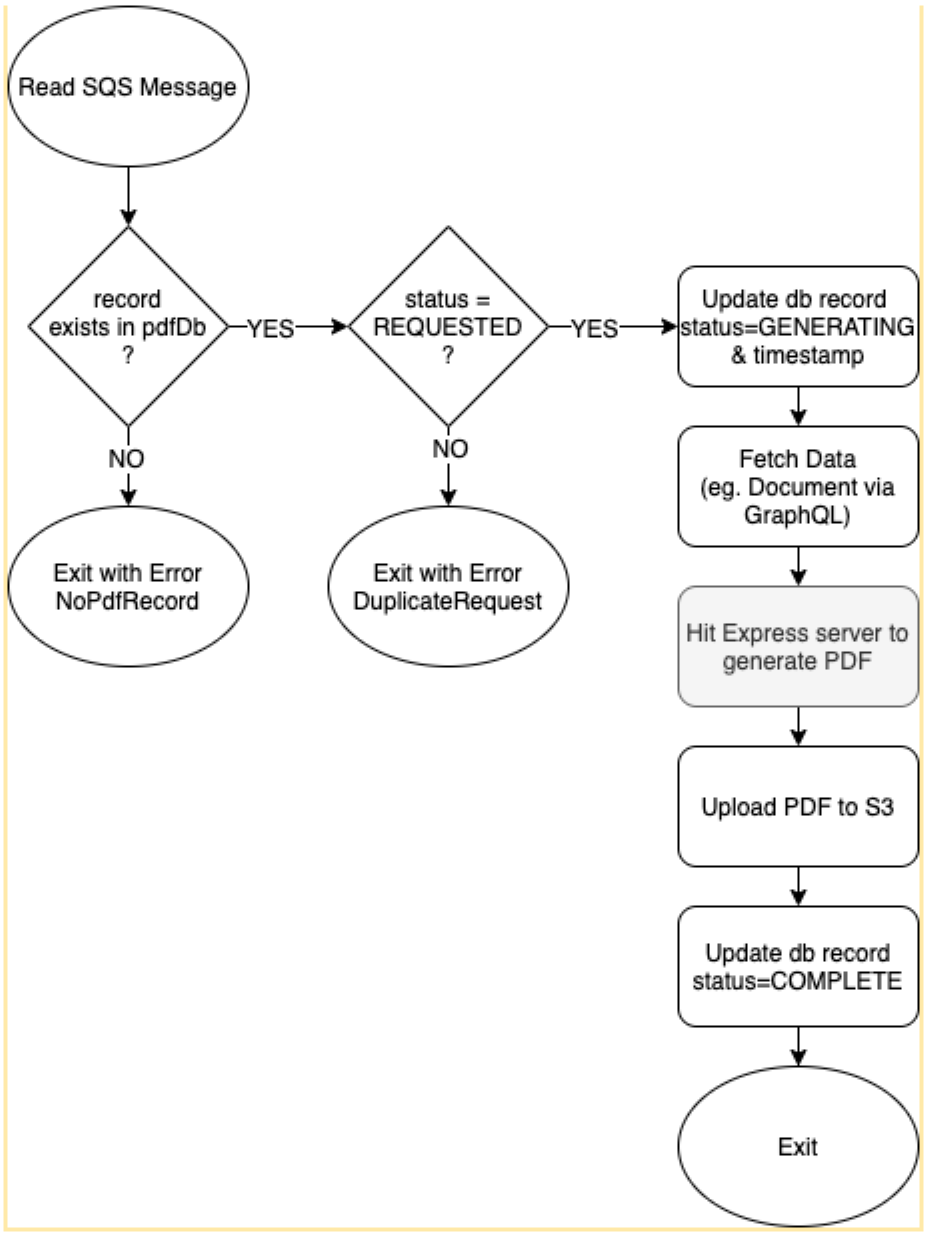

Mutation: GeneratePdf
args:
- org_id
- document_uuid
PDF for report exits in db?
YES
Use report_type to invoke correct cache handler
Cached PDF is up-to-date?
YES
status = COMPLETE ?
YES
Return PDF pre-signed URL
NO
NO
NO
Return Status Message
Create db record status=REQUESTED
Add message to SQS
Example cache handler for driver_documents
Fetch document_updated via GraphQL
report_created > document_updated

PDF Server Program Flow (Create Mutation)

SQS Queue - Since the PDF generation process could be long running, it should be done asynchronously via a queue and worker. The message sent to the queue will be a UUID that references an entry in the pdf database.
+Samsara SQS

PDF Renderer Worker -
Go SQS worker that runs react-pdf script to generate PDF. Handles setting generation status and uploading to S3.

The worker will need to reach out to documents service to gather the document data for rendering.


Read SQS Message
record exists in pdfDb ?
YES
status = REQUESTED ?
YES
Update db record status=GENERATING & timestamp
NO
Exit with Error NoPdfRecord
NO
Exit with Error DuplicateRequest
Fetch Data (eg. Document via GraphQL)
Hit Express server to generate PDF
Upload PDF to S3
Update db record status=COMPLETE
Exit

PDF Renderer Worker Program Flow

Node PDF Renderer
The emailrenderer service (a node/express server) will be extended with an endpoint that accepts a json of document data and streams back a PDF of the document. The renderer will use react-pdf.

React-pdf allows for creating PDF templates at a high level using familiar React syntax. Compared to other approaches this should allow us to build out rendering capabilities for various report types with minimal engineering effort.

EmailRenderWorker

PDFdb - Contains data about the PDF and it’s generation status.

```
SCHEMA:


TABLE: pdfs


org_id: int
uuid: binary(16) - unique identifier for a pdf
user_id: int
s3_bucket_name: string
s3_object_key: string
document_uuid: string - used to identify data for a report and
cacheing
~~report_type: enum - [DOCUMENT, ...]~~
pdf_last_generated_ms: int KEY2 - timestamp for when pdf was g
enerated
status: enum - [REQUESTED, GENERATING, COMPLETE]


PRIMARY KEY (`uuid`)
```

Alternatives Considered

- Use a client side generation library:

PROBLEM: this doesn’t allow for the flexibility of uses that server side generation allows, such as via Kondo API, event driven, etc..

- Use a go library for PDF generation server side:

PROBLEM: Existing libraries are too low-level and would require excessive engineering effort to get template layout and formatting right for different use cases.

- hybrid go/node worker environment

    - See +Dev Spec: Video Download Enhancements for an example of calling a script from a go worker. Relevant code: https://github.com/samsara-dev/backend/blob/eea9f9e52946b325281d68434c90dbe3e6bdc1a8/go/src/samsaradev.io/fleet/safety/infra/videodownloadworker/overlay.go#L115
    - Dockerfile used by videodownloadworker: https://github.com/samsara-dev/backend/blob/master/docker/python-ml/Dockerfile

Open Questions

- How will we handle the dynamic nature of PDF layout/rendering?

  > *dynamic layout is tricky because usually they have to render to a page size, so it makes a lot of the logic difficult where on things like web and mobile you can scroll forever and don't have to think about pagebreaks*

  Pagebreaks are a great example of a problem that is somewhat unique to PDF rendering. A lot of lower level libraries provide the primitives to construct a PDF document, but leave the calculations and decisions about page breaks in the hands of the engineer 😞 💸. Other libraries simplify their api by scrapping html pages and making all the decisions for you, resulting in images, tables, etc. broken up into different pages arbitrarily.

  React-PDF falls right in the middle, doing the calculations for the developer while also allowing for control via breakable/unbreakable components. It can also do hyphenation at linebreaks.

- How do we make creating visually appealing reports simple yet flexible, and able to meet the majority of our use cases?

  > *there aren't off the shelf versions where someone is like "oh every company has this problem you end up using this package and it will work". Every enterprise software company wants this, we're not the only ones to wish for printable pdf reports.*

  > *the requirements for pdfs are very different across applications (I want a pdf version of this report versus I want a pdf with one image on each page) and don't end up being re-usable*

The goal here is finding something that is flexible in the documents it can create, yet requires an acceptable amount of engineering effort to author a new report template. Low-level libraries fail on the “amount of engineering effort” category and often lead to developers creating bespoke design-systems over the crude api, adding additional complexity and losing flexibility. High-level libraries fail on the “flexible in the documents it can create” category, offering to render any html page you can point them to, but often rendering a broken mess without much recourse.

React-PDF falls in the middle of this as well, providing familiar paradigms like React, JSX and CSS so that most engineers will be comfortable authoring reports in a reasonable amount of time. At the same time, there are a few specialized components and some advanced levers that help solve common issues when dealing with PDF rendering. Try out the REPL and see for yourself!

- Can we support rendering SVG images into the PDF?
  React-PDF supports canvas drawing and there are examples of drawing svg to canvas.

Milestones

Timeline			
Title	**Dates**	**Assigned to**	**Description**
⍰ On-call	Mar 10, 2020 - Mar 16, 2020	Shaun H	
⍰ Milestone 1 (node app)	Mar 11, 2020 - Mar 17, 2020	Noah R	
⍰ Milestone 2 (infra)	Mar 11, 2020 - Mar 13, 2020	Ashcon Z	
⍰ Vacation	Mar 16, 2020 - Mar 16, 2020	Ashcon Z	
⍰ Milestone 2 (infra)	Mar 17, 2020 - Mar 18, 2020	Shaun H	

On-call	Mar 17, 2020 - Mar 23, 2020	Ashcon Z	
⍰ Milestone 5 (beautification)	Mar 18, 2020 - Mar 24, 2020	Noah R	
⍰ Milestone 2 (go worker)	Mar 19, 2020 - Mar 27, 2020	Shaun H, Ashcon Z	
⍰ Milestone 3 (pdf server)	Mar 24, 2020 - Mar 30, 2020	Ashcon Z, Shaun H	
On-call	Mar 25, 2020 - Mar 30, 2020	Noah R	
⍰ Milestone 4 (kondo)	Mar 30, 2020 - Apr 3, 2020	Shaun H	
Milestone 6 (etc)	Apr 6, 2020 - Apr 17, 2020		

JIRA EPIC:

https://samsaradev.atlassian.net/browse/DM-708

✅ Milestone 1 - Create react-pdf proof of concept application (5 days)

Create server-side node app that takes document JSON and generates PDF. This proves out the document PDF generation concept using react-pdf.

- https://react-pdf.org/
- Simple react-pdf app - https://gist.github.com/Shaun1/2d0b375bb7ce66994b5209439378f62b
- Starting point for package.json - https://gist.github.com/Shaun1/d3f0baa520d14e573032f7c7e9a93ae9
    - ☑ DM-709 ~~- Create simple node script for rendering PDF based on data input and template~~
    - Create simple template for driver document PDFs

✅ Milestone 2 - Set up go worker and node pdf-generation service (15 days)

- +How to: Create a new SQS Service

Create a go based SQS consumer that invokes the PDF Renderer process.

- ☑ ~~DM-710 - Create SQS queue @Ashcon Z~~
- ☑ ~~DM-715 - Set up SQS dead letter queue~~
- ☑ ~~DM-711 - Create SamsaraDocumentPdfs S3 Bucket~~
- ☑ ~~DM-713 - Create document_pdfs table~~
- ☑ ~~DM-712 - Create DocumentPdfRenderer SQS worker~~
- ☑ ~~DM-714 - Modify documentpdfworker accommodate react-pdf~~
- ☑ ~~DM-746 - Create PDF model bridge and methods~~
- ☑ ~~DM-747 - Create PDF proto model~~
- ☑ ~~DM-716 - Add golang worker logic for message handling, dead letter, script, logging, etc~~

✅ Milestone 3 - Add PDF endpoints to Documents Server (5 days)

- ☑ ~~DM-768 - Create API proposal and solicit feedback from API team~~
- ☑ ~~DM-759 - Create grpc endpoint for requesting pdf creation~~
- ☑ ~~DM-761 - Create field func for pdf creation request~~

✅ Milestone 4 -Add Kondo API endpoint (5 days)

- + Making the move to Kondo
- ☑ ~~DM-762 - Create kondo endpoint and registry entry which uses PDF server graphQL endpoint~~

✅ Milestone 5 - Refine PDF output quality to meet customer standards (5 days)

- ☑ ~~Make sure margins, fonts, images, page breaks etc. meet design standards~~
- ☑ ~~Test edge cases with different types of documents~~

Milestone 6 - Iterate on PDF design quality, instrumentation, reliability, localization, etc (10 days)

- [ ] Bug and enhancement tickets are being tracked in this epic https://samsaradev.atlassian.net/browse/DM-708
- [ ] Test Plan: +PDF Generation Test Plan

Deployment / Rollout Plan

- Service will be fully rolled out initially for internal use only
- Conduct testing / load testing by hitting graphQL endpoint
- Standard permissions checks take place using go libraries
- Kondo API roll out??

Monitoring, Maintenance, Metrics

- [ ] Number/Frequency of PDFs generated by org, document type
- [ ] Generation time
- [ ] Error logging (sentry)

References

+PRD: Document PDF Generation
+[Design Spec] PDF Download- 07/19
+Discussion on Chart Rendering with Edwin
+ RFC: "Email Myself"
+Samsara SQS
+PDF Export Template
+PDF Generation Test Plan