# 

Engrafa Document Manager Application Design Specification

|  |  |  |
| --- | --- | --- |
| **Author** | **Revision Number** | **Date** |
| Architect | 1.0 | October 9, 2014 |
|  |  |  |
|  |  |  |

Engrafa Document Manager Application Design Specification 1

Application Design Specification 3

1. Design 3

1.1 Work Flow Description 3

1.1.1 Document Manager Worker Execution 3

1.1.2 View Document as HTML 4

1.1.3 Process Document 5

1.1.4 Notes for Initial Submission Feedback 5

1.2 Component Requirements 5

1.2.1 Custom Software Components 5

1.2.2 TopCoder Generic Components 6

1.2.3 Third Party Components 6

1.2.4 Assemblies 6

1.3 Application Management 6

1.3.1 Transaction 6

1.3.2 Configuration 7

1.3.3 Persistence 7

1.3.4 Thread-Safety and Concurrency 7

1.3.5 Logging 7

1.3.6 Auditing 7

1.3.7 Exception Handling 7

1.3.8 Internationalization 7

1.3.9 Security 7

1.3.10 Performance 8

1.3.11 Scalability 8

1.4 Deployment Constraints 8

1.4.1 Technology overview 8

1.5 Development Standards: 8

1.6 Interfaces Classes Overview 9

1.7 Changes to Existing System 9

2. User Interface 9

3. Included Documentation 9

3.1 Architecture Documentation 9

4. Future Enhancements 9

# Application Design Specification

# Design

Currently, the Engrafa Service exists as the Workflow Manager. At high level, the service pulls documents from cloud storage ([Box](https://www.box.com)) and moves them through a series of tasks on a cloud work platform ([CrowdFlower](http://www.crowdflower.com)) to crowd source data extraction. Once each task in the flow has completed, the Workflow Manager aggregates the results into a single structured data file (XML), back to the cloud storage.

The Document Managerwill facilitate uploads and downloads of documents to and from the cloud storage. This responsibility will no longer be handled by the Workflow Manager. Additionally, the Document Manager will use a service ([CVISION Trapeze](http://www.cvisiontech.com/trapeze/general/trapeze.html?lang=eng)) to compress, carry out skew correction, and improve OCR for the documents. Document Manager will leverage Box’s View API to convert PDFs to HTML5 for additional viewing flexibility.

This initial version will be headless (no GUI), but may be extended later to add a frontend.

## Work Flow Description

This application consists of two sub-modules - Document Manager and Document Processor.

* Document Manager will be deployed on AWS EC2 instance and will be responsible for retrieving documents from monitored client Box folders, passing the documents Document Processor for processing and converting the processed documents to HTML5 using Box View API.

Document Manager will consist of two separate processes - web server process and worker process. The web server process will provide one web service for viewing document as HTML. The worker process will execute various tasks periodically to fulfill the required processing.

* There will be one or more Document Processors, each will be deployed on an AWS EC2 instance. Document Processor will use CVISION Trapeze command line to process documents. Document Manager will dynamically launch/start/stop Document Processors (using AWS EC2 web services) according to its workload.

One Document Processor will consist of one web server process, which will provide web service for processing document posted to the web service and retrieving CVISION Trapeze license usage log.

### Document Manager Worker Execution

Engrafa Document Manager will periodically do the following tasks separately:

* Scan a collection of pre-configured Client Folders, and check for new input files in Box, for each new file, create a Document record and en-queue the record - the business logic is mainly implemented in DocumentService.pullNewDocuments and DocumentService.enqueueDocument
* Process Document records with "QUEUED" status – the business logic is mainly implemented in DocumentService.processDocuments and DocumentService.processDocument
  + Retrieve all "QUEUED" Document records
  + For each "QUEUED" Document record
    - Update Document record status to "PROCESSING"
    - Choose the least-busy Document Processor and update its workload
    - Download the Document file from Box
    - Send the Document file to the chosen Document Processor to process the document (via Document Processor web service)
    - Upload the Document processing result to Box
    - Update Document record status to "PROCESSED"
    - Update the Document Processor's workload
    - Remove original file from Box and temporary files in local file system
* Process Document records with "PROCESSED" status – the business logic is mainly implemented in DocumentService.convertDocuments and DocumentService.convertDocumentToHTML
  + Retrieve all "PROCESSED" Document records
  + For each "PROCESSED" Document record
    - Update Document record status to "CONVERTING"
    - Post document URL to Box View API for conversion
    - Save the Box View API document ID in the Document record
* Process Document records with "CONVERTING" status – the business logic is mainly implemented in DocumentService.checkDocumentConversionStatus
  + Retrieve all "CONVERTING" Document records
  + For each "CONVERTING" Document record
    - Update Document record status to "CONVERTING"
    - Check document conversion status using Box View API
    - If the conversion is done, update Document record status to "CONVERTED"
* Scale Document Processors - this business logic is mainly implemented in DocumentService.scaleDocumentProcessors
  + Find Document Processors that have been idled for max-allowed time, and stop the AWS EC2 instances.
  + Find Document Processors that have been idled for max-allowed time for termination, and terminate the AWS EC2 instances.
  + Check if either or both of the following two conditions is met:
    - The number of Document records in "QUEUED" status exceeds the configured threshold
    - Any Document record has been in "QUEUED" status for more than a configured threshold
  + If either or both of the above-mentioned conditions is met, then another Document Processor will be needed
    - If there're stopped Document Processors in the system, then start one Document Processor
    - If there is no stopped Document Processors in the system, then launch a new Document Processor from configured AMI.
* Refresh Box Access Token - this business logic is mainly implemented in config.refreshBoxAccessToken.
* Pull Latest Trapeze License Usage Log - this business logic is mainly implemented in DocumentService.pullTrapezeLicenseUsageLogs.

### View Document as HTML

Document Manager will provide a web service for viewing a document as HTML. The web service will simply create a view session with Box View API, extract the viewer URL and redirect to the Box View API viewer URL for the document. The viewer URL can be displayed using [Box View API viewer](https://github.com/box/viewer.js).

The requirement document raises a question regarding whether to store and when to do the conversion.

In the proposed architecture, the conversion result is not stored separately (local or in Box) other than the Box View API document. Keeping a separate copy of the conversion result is not necessary because Box View API already stores the conversion result and provides service to view the converted HTML content directly.

The conversion is done after a document is processed, and any requests made to view the document as HTML before the conversion completion will be rejected. However, the PDF version of the processed document is still available, therefore the consuming frontend pages will have an option to try to render HTML version first, and fall back to PDF version if HTML version is not available yet.

### Process Document

Document Processor will provide a web service for processing a document. The web service will accept the document in HTTP POST request, process the document using pre-configured CVISION Trapeze command (with tuned options for processing) and finally send back the processed document.

Document Processor will use [shelljs](https://www.npmjs.org/package/shelljs) library to execute the CVISION Trapeze in a separate process.

The CVISION Trapeze command for document processing will look like the following (options for compression, skew correction and OCR enhancement are included):

"C:\Program Files (x86)\CVision\Trapeze 1.2\CVTrapeze.exe" -m 1 -c ON -acroversion 0 -colorcomptype 2 -mrcquality 7 -mrcColorCompType 0 -mrcresample 1 -mrcSegType 2 -mrcPRed 2 -mrcPDCTQuality 40 -linearize -o -oocr -ocrmode accurate -okeepname -qualityc 75 -qualityg 75 -rscdwndpi 300 -rscinterp smartbicubic -rsgdwndpi 300 -rsginterp smartbicubic -rsbdwndpi 300 -rsbinterp smartbicubic -cconc -ccong -config "<CONFIG>" -redirstderr -in <INPUT\_FILE> -out <OUTPUT\_FILE>

### Notes for Initial Submission Feedback

* AWS SDK

*We were hoping for more details around how this will be used, either direction in the Application Design Specification, or in the related Assembly Specification documents. We think it would be useful for someone competing in the next Assembly challenge(s) to have more details on which APIs they should use. Is there a place for that within the current deliverables?*

AWS SDK usage is very detailed in TCUML method implementation of DocumentService#scaleDocumentProcessors.

* Trapeze

*It might be nice to include more details around how the Document Manager will interact with this process. Since a document can take a while to process, should there be any way to monitor the Trapeze process? How should errors be handled?*

This is detailed in "1.1.3 Process Document" section, and also the implementation notes of DocumentProcessor#processDocument. As per requirement, a typical document takes 10 seconds to process. Each document processing session will be an HTTP request, and at Document Manager side, the HTTP request will use [agentkeepalive](https://www.npmjs.org/package/agentkeepalive) to make the HTTP request timeout large enough for typical document processing.

* Separation of Responsibilities

*We originally were envisioning a main Document Manager controller that would be responsible for managing AMIs as well as Trapeze processes for each. We understand there may be reasons to include additional modules/controllers, or additional layers to the application, to help separate responsibilities within the application. If you are adding more layers or modules, would you be able to provide background/detail around this decision, perhaps in the Application Design Specification? This will help us understand the complexities, but also help us understand if we will need to make any adjustments to our project timeline and/or budget.*

The architecture layering and workflows are explained in section 1.1, if you need more information on a particular topic, please let me know.

## Component Requirements

### Custom Software Components

#### *New Custom Components*

None

### TopCoder Generic Components

None

### Third Party Components

* [Node.js 0.10.29](http://nodejs.org) (MIT License)

This application is built on Node.js platform.

* [Express 4.9.7](http://expressjs.com) (MIT License)

This is used as the web framework.

* [MongoDB 2.6](http://www.mongodb.org) (Creative Commons Attribution 3.0 Unported License)

This is the database system used in this application.

* [Mongoose.js 3.8.12](https://www.npmjs.org/package/mongoose) (MIT License)

This is used to simplify the interaction with MongoDB.

* [winston 0.7.3](https://www.npmjs.org/package/winston) (MIT License)

This is used for logging.

* [superagent 0.18.1](https://www.npmjs.org/package/superagent) (MIT License)

This is used to access Box REST APIs.

* [underscore 1.6.0](https://www.npmjs.org/package/underscore) (MIT License)

This is used as utility for various functional programming capabilities such as iteration, filtering and join.

* [forever 0.11.1](https://www.npmjs.org/package/forever) (MIT License)

This is used to start the application as a background process.

* [async 0.9.0](https://www.npmjs.org/package/async) (MIT License)

This is used to simplify asynchronous programming.

* [aws-sdk 2.0.19](https://www.npmjs.org/package/aws-sdk) (Apache License 2)

This is used to access Amazon Web Services.

* [shelljs 0.3.0](https://www.npmjs.org/package/shelljs) (BSD License)

This is used to execute shell commands for running CVISION Trapeze.

* [uuid 2.0.1](https://www.npmjs.org/package/uuid) (MIT License)

This is used to generate UUIDs.

* [agentkeepalive 1.2.0](https://www.npmjs.org/package/agentkeepalive) (MIT License)

This is used to issue keep-alive HTTP requests.

### Assemblies

* CrowdReason Engrafa Document Processor Assembly

This assembly will implement the Document Processor and Document Processor Scaling logic in Document Manager.

* CrowdReason Engrafa Document Manager Assembly

This assembly will implement the Document Manager.

## Application Management

### Transaction

MongoDB doesn't support complex transaction management like conventional RDBMS, but it does support [atomic operations on single documents](http://www.mongodb.org/display/DOCS/Atomic+Operations), which can guarantee atomic operations on single nested objects (aka "documents" in MongoDB terminology), and that's sufficient for this project.

This application will interact with Box, AWS and Document Processors through REST APIs, and multiple REST API calls and database operations would be involved when processing a single input PDF file (downloading file from Box, creating Document record, deleting file from Box, uploading file to Box, etc.), it is not easy to achieve all-or-nothing transactional behavior, therefore it is crucial to include detailed information in error logging.

### Configuration

The configurations will be specified in JavaScript file config.js, all configurations are clearly shown on "Document Manager Class Diagram" / "Document Processor Class Diagram" and all configuration options are required.

JavaScript services and main application script will import the config.js as a Node.js module using "require" statement.

Client folder configurations are stored in MongoDB collection ClientFolder.

### Persistence

Application data are stored in the [MongoDB](http://www.mongodb.org), and [mongoose](http://mongoosejs.com) is used as the ODM library for easy MongoDB access.

The input and output files are stored in [Box](https://www.box.com) private folders. [Box Content API](https://developers.box.com/docs/) will be used to access files stored in [Box](https://www.box.com).

Box access and refresh tokens will be saved to local file system as text file when refreshed.

### Thread-Safety and Concurrency

Applications built with JavaScript and Node.js is inherently asynchronous, callbacks(JavaScript) are used extensively to coordinate asynchronous tasks/operations. There's generally no thread safety concern with Node.js powered applications.

Document Manager worker.js will schedule several tasks to be executed periodically (with setTimeout function), but all tasks are in the same event-loop (of the same thread) since a Node.js application is inherently single-threaded.

### Logging

The JavaScript code will perform logging using [winston](https://www.npmjs.org/package/winston) Node.js module.

Errors will be logged at ERROR level, debug information will be logged at DEBUG level.

### Auditing

There is no auditing requirement.

### Exception Handling

The functions of JavaScript services are implemented in asynchronous manner, i.e. virtually all functions will take a callback function that will be called to notify function caller of result. If error occurs, callback function will be called with a String type parameter "error" detailing the error.

### Internationalization

There is no internationalization requirement.

### Security

[Box Content API](https://developers.box.com/docs/) requires API client to include OAuth2 access token in "Authorization: Bearer " HTTP header, and the OAuth2 access token will be configured in config.js. The access token and refresh token will be refreshed periodically and once refreshed, they will be stored in local cache file.

Communications with Box View API, Box Content API and AWS will be through HTTPS.

AWS credentials should be configured according to [AWS JavaScript SDK guideline](http://docs.aws.amazon.com/AWSJavaScriptSDK/guide/node-configuring.html#Setting_AWS_Credentials).

The AWS EC2 instances for Document Processors should use [Security Groups](http://docs.aws.amazon.com/AmazonVPC/latest/UserGuide/VPC_SecurityGroups.html) to restrict ingress requests (to the Document Processing web service) to whitelisted IPs. This can be configured with the AMI image.

### Performance

Mongoose internally uses a connection pool to improve performance, therefore it is not necessary for this application to manage connection pool by itself.

The most time consuming and computation intensive task is document processing, and this is done using a pool of Document Processor instances. The Document Processor pool will be scaled automatically, so that new Document Processor instances will be launched when the queued document increases and existing Document Processor instances will be stopped/terminated when idled for long time.

### Scalability

As explained in 1.3.10, the Document Processor pool will grow or shrink automatically according to the workload of the application.

## Deployment Constraints

Both Document Manager and Document Processor will be deployed on Amazon EC2 instances, and Document Processor must be deployed on Windows-based EC2 instances because CVISION Trapeze requires Windows platform.

Document Manager will consist of two separate processes - web server process and worker process. The web server process will be an Express-powered web application. The worker process will be started using Node.js [forever](https://www.npmjs.org/package/forever) module so that it will be started as a background process. Please refer to [forever documentation](https://github.com/nodejitsu/forever) for specific commands to start a script.

Document Manager will use the MongoDB to store the data.

Document Manager will access Box Content API and Box View API.

Document Manager will access AWS to manage EC2 instances.

Document Processor will execute CVISION Trapeze to process documents.

### Technology overview

* JavaScript
* [Node.js 0.10.29](http://nodejs.org)
* [Express 4.9.7](http://expressjs.com)
* [MongoDB 2.6](http://www.mongodb.org)
* [Mongoose.js 3.8.12](https://www.npmjs.org/package/mongoose)
* [winston 0.7.3](https://www.npmjs.org/package/winston)
* [superagent 0.18.1](https://www.npmjs.org/package/superagent)
* [underscore 1.6.0](https://www.npmjs.org/package/underscore)
* [forever 0.11.1](https://www.npmjs.org/package/forever)
* [aws-sdk 2.0.19](https://www.npmjs.org/package/aws-sdk)
* [shelljs 0.3.0](https://www.npmjs.org/package/shelljs)
* [uuid 2.0.1](https://www.npmjs.org/package/uuid)
* [async 0.9.0](https://www.npmjs.org/package/async)
* [Box Content API](https://developers.box.com/docs/)
* [Box View API](https://developers.box.com/view/)
* [Amazon Web Services](http://aws.amazon.com)
* [Amazon EC2](http://aws.amazon.com/ec2/)
* [CVISION Trapeze](http://www.cvisiontech.com/trapeze/general/trapeze.html?lang=eng)

## Development Standards:

The assembly development must adhere to the guidelines as outlined in the [TopCoder Assembly Competition Tutorial](http://apps.topcoder.com/wiki/display/tc/Assembly+Competition+Tutorials).

## Interfaces Classes Overview

See the TCUML file.

## Changes to Existing System

None

# User Interface

This initial version will be headless (no GUI), but may be extended later to add a frontend.

# Included Documentation

## Architecture Documentation

* Class Diagrams
* Sequence Diagrams
* Application Design Specification
* Assembly Specifications

# Future Enhancements

None