Overview of the project

The high level view of the project is to provide an application for easily, efficiently, and securely converting a client's paper forms into an electronic copy, stored in a local database. For example, imagine a dentist's office; forms for health information, liability waiving, etc. are given to the patient to fill out at the first appointment. These forms end up having to either be manually entered into a system, or referenced directly (and stored in a file cabinet) from that point on. This project would offer the ability to pass the document through a scanner, the output of which the proposed application will convert into an electronic copy. This is then stored in the office's document database, which is provided by the application. If relevant, it would also be possible to index the document in the database by the contents of the form.

This is achievable using standard image processing techniques, document/field tagging and optical character recognition (OCR. The objective, formally, is to be able to quickly scan through (potentially) unordered stacks of documents, parsing out a client's input, and storing them as an atomic object in a database identified by a unique id, and any keyed form data fields. In order to do this, first a user would design a form document - specific for their application - which would encode document identifiers, page orderings, and so on in a QR code on a nondescript part of the page. The locations of input elements on the designed form would be stored in the application to allow for later extraction of a client's written input using OCR. The retrieved data, identified by the QR code, can then be stored to a database. This allows for either the raw data to be referenced by another software solution, or to allow for the document to be re-rendered as a pdf with selectable text as opposed to the raster produced by standard scan operations. The paper documents could then be shredded and recycled, allowing for potentially less paper having to be created.

Project Architecture

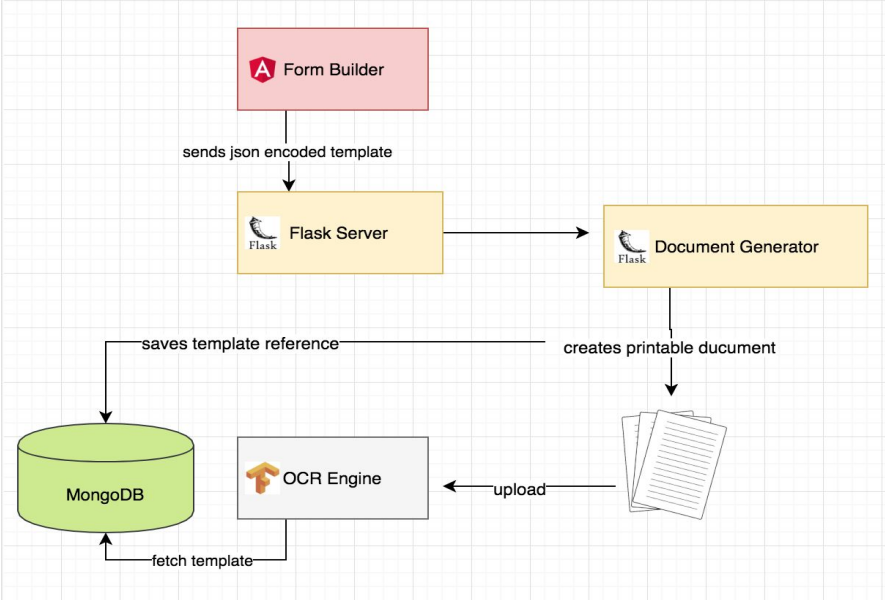


Figure 1.1

FormBuilder

The FormBuilder on top of the Angular 4 framework. The builder should have a side panel and the form content region. The side panel consists a list draggable form elements. As of now, we have three type form element implemented, which includes text field, title view, and column view. Whenever an element is dragged inside the content region, a new element (based on the type) is created and inserted into the form region. Until form fully completed, each element is stored at local data manager. When the user sends a post request via submit, the data manager bundles the active form into a template and sends to server.

Document Generator

The pdf generator is written with pyfpdf, a framework for creating pdf files. Using fpdf objects such as rectangle, line, text, and images, our encoded digital form was transcribed and recreated in exact precision using these drawable object. Each generated document is added to the database for later reference.

New Patient Form

78e28564-0ca9-48fa-82d6-1885669e383b

Patient information

Address

555 Huntington Ave

Please fill completely

First Name

Rixing

Last Name

Wu

Where did you hear about us?

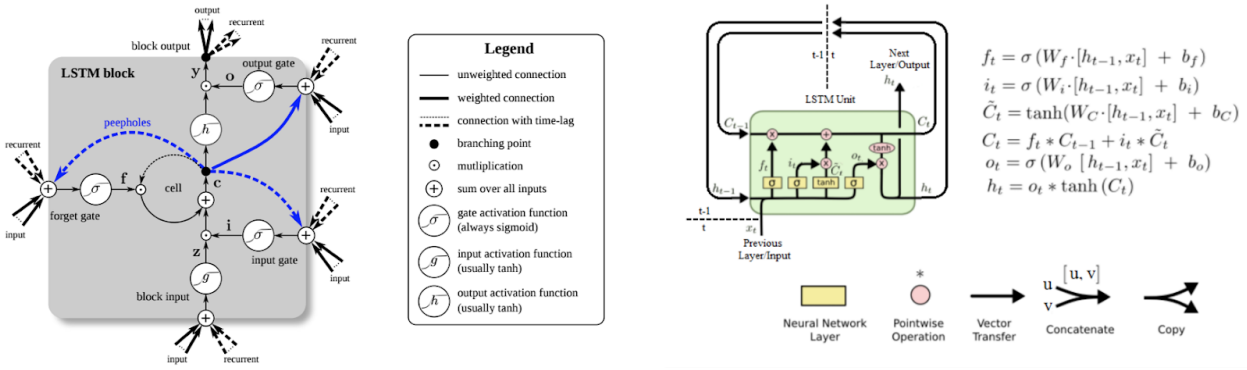
Friends

An example of a filled form which generated with by Document Generator

Database

Initially, we wanted DocuScan to operate like Saas and having users and profiles was part of our solution, so a database was needed. However, the implementation of a custom OCR engine has taken much more of our time than expect as the Tesseract OCR did not work out in our favor. So, less important features such as accounts and profiles has been abandoned. For our proof of concept application, we have implemented a single database consisting a table of form objects, which can be accessed with a universally unique identifiers.

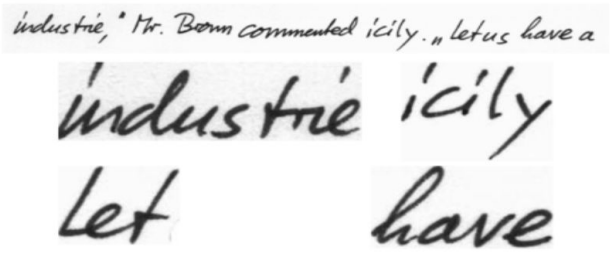
OCR Engine



Images that are to be processed by the network (either during training or evaluation) will first pass through a series of transforms: skew reduction, batch normalization, and zero-padding on the image edges to ensure that the 2D sub-sampling operation does not hit any index errors - but also does not introduce irrelevant data into the example. The network architecture is that of an inverted pyramid. Initially, images are broken up into small “context regions” (really just “rectangular subsets” of the data’s dimension, e.g. an image gets broken up into small 2d blocks of pixels), which are then passed to the first layer of MDLSTM’s to be scanned in all relevant directions (the order of the sequence passed to the LSTM): top-right->bottom-left, top-left->bottom-right, bottom-right->top-left, bottom-left->top-right. The directional traversals happen in parallel, as part of a single “layer”, whose activations get summed into a single output vector. That output is then passed to a standard feedforward layer which sums over the outputs from the distinct directional layers, collapsing them into a single output vector. we’ve used the term MDLSTM here under the expectation that the final network will use said methodology as the final computational work-horse. However, currently, only the MDRNN version is working (the difference being that LSTM are gated whereas RNN’s

are not) - the LSTM version requires so many parameters, and is quite hard to debug - so in truth we was expecting a bit more of an implementation challenge on that one. An example of training data can be seemed in figure 1.2, provided by Bern University Research Lab.

Figure 1.2



Problems

In order to crop data from the scanned pdf later, the exact position and dimension of text box were recorded. What we did not account for was the variation of paper size as users might print their forms in A4, A3, A2 or letter size, which is problematic for cropping. Our proposed solution was to add markers to the four corners of the pdf, so when the form is scanned, the position of each textbox can be computed relatively to the four markers on the scanned document. To get the position of the markers, OpenCV was used to detect the square marker positions. As you can see in figure 1.3, the top left marker was detected, however, it was also picking up unexpected markers in the QRCode.

figure 1.3

Goal and Outcome

Our initial goal was to build a formbuilder to design templates and the OCR engine to process handwritten characters. For the formbuilder, most features we wanted has met, such as the ability to drag and drop, edit, and save and load template. Built using angular cli front-end framework, the formbuilder can run independently as a standalone web app. For the backend, PDF objects could be generated from an encoded json. Coming in to this project, we expected the project might not succeed because handwritten characters recognition is difficult and currently no out of box solution exists in the market. Google's Tesseract OCR works well with printed characters, but not handwritten. So we decided to implement our own solution. An LSTM neural network for partially constrained handwriting recognition. Our implementation was about to predict more than two-thirds input data that it was given.