Kameron Weaver
Skylar Lowery
Minh Trung Pham
Full Contact
6/19/2012

Introduction

Founded by Bart Lorang, Travis Todd, and Dan Lynn, Full Contact brings together contact data from different sources (phone, email, facebook, etc.) and compiles them in one comprehensive location. The company focuses on building fresh, complete, and accurate contacts, then further enriching them with pictures, social links, and more.

Full Contact's API utilizes powerful graph based approaches to create a single complete contact data model. Fragmented data is collected, normalized, then combined to become a complete contact. This contact is then enriched with information from public social profiles like Facebook and Twitter to provide social media handles and addresses. To provide further support, Full Contact utilizes similar technology to update current contacts and push the updates across all platforms providing for a single unified data set. Not only is contact data full and detailed, but up to date and easily usable - all in one place.

As an extension to Full Contact's API, our task was to develop an API in which business cards could be used to populate contact information. The project's focus was in taking a business card, reading the data, then constructing a contact data model that was consistent with Full Contact's existing model. The resulting contact model could then be easily integrated with any existing Full Contact service to update a user's address book.

Product Vision

The vision of the image processing API is to give the Full Contact API and application users the ability to transform these business cards, and more so any images, into digital contact data. The most critical aspect of this product is accuracy as it is the main selling point of the product. The Full Contact Data Model is also a huge advantage that this image transcription provides. The Full Contact Data Model can be integrated with all of Full Contact's services including enrichment and de-duplication. Input diversity is also a huge advantage. What makes this unique compared to typical solutions is that it offers the ability to transcribe any image. This product solves the problem of "What do I do with all these business cards" while also making it possible to transcribe signs, pictures, and anything you can take a picture of that has relevant contact data on it.

Requirements

Functional requirements

- The API Solution must accept a large array of image file types
    - Being that end-users will capture and obtain their own images however they can, the image transcription task needs to be usable with a variety of image formats
- Images used as input should be stored to be fetched later
    - The burden of storing the initial input image should not be placed upon the End user. Since the provided image may be used as a part of the contact model, it must be stored in some central location

- End-User should be able to submit multiple jobs
    - the solution should be able to handle multiple requests from a single user, as well as multiple users. There should not be a limitation to how many cards can be processed at a single time.

- End-User should be able to provide feedback regarding quality / accuracy of results
    - the end user should be provided opportunity to give feedback regarding quality of the transcription. For the sake of quality control, this feedback may be used to modify the transcription process for better results.

- Returned results can be accepted by the end user
    - Before the results are added to the end-users cloud address book, the user must be given an opportunity to preview and accept the results as passable.

- Returned results can be rejected by the end user
    - Before the results are added to the end-users cloud address book, the user must be given an opportunity to preview and reject the results, in the event that the transcription process gave lacking results.

- Image transcription should work for two-sided Business cards as well as one sided ones
    - business cards often contain relevant information on more than one side. It is crucial that the application should be able to process both sides of one card into a single contact model.

Non-Functional requirements

- API-based solution must follow REST methodology

- Solution must be scalable

- Solution must be accessible by many simultaneous end-users

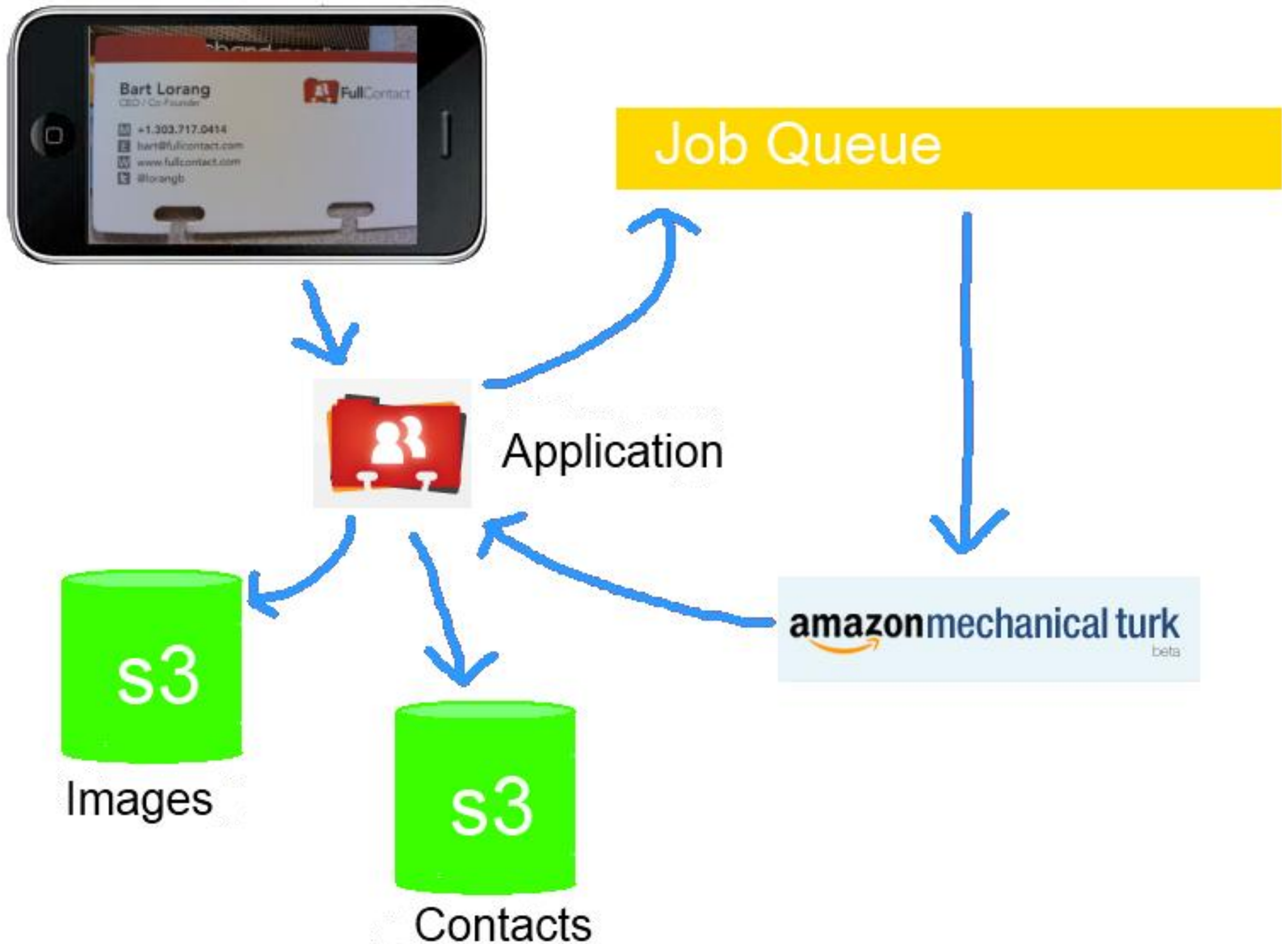
- Returned results should arrive in a reasonable amount of time (minutes)

- Results must be accurate (~99%)

- User involvement should be kept to a minimal

System Architecture

Basic Process



API Endpoint

The user sends an image to the application with metadata including an API key to the application. The application stores the image(s) on Amazon S3 Cloud Storage, takes the URL to this image and the metadata, and places it into a job queue. The application reads from the job queue and creates Human Intelligence Tasks for Amazon Mechanical Turk known as a HIT. These HITs are created and maintained using HTTP requests. When the HIT is completed on mechanical turk the data is pulled from Turk and modeled as a Full Contact contact model. This contact is saved to S3 in a folder named by the API key. The final stage is storing the contact model data file in the cloud.

Amazon Services

The application is very heavily integrated with numerous Amazon services. The S3 Simple Storage Service is the data store for the final resulting contact data files, the images that are being processed, and for web hosting the Mechanical Turk iFrame.

Simple Storage Service

Amazon's Simple Storage Service provided valuable, reliable data storage. It is not labeled as a relational data store and works similarly to a NoSQL datastore like MongoDB. We used a Java/Groovy library to communicate with the database. Anything, down to byte code, can be stored on S3 by creating "PutObjectRequests" and shipping them off to the database. The Object Requests contain a key value, an object to store, and many other optional meta data such as file permissions. One great feature of S3 is its ability to make temporary URL's for objects in the database for hidden files which can be useful depending on future wants of the API. For example Contact data may be public, hidden, or temporarily viewable only by those with specific URL's (which is most likely the case) in the future.

Simple Message Queue

Amazon's Simple Message Queue was used to store requests awaiting transcription. Since completed data is temporarily stored on Mechanical Turk, all that was needed was to store metadata that made communication with all our other resources possible while minimizing extra disk usage of any kind. The Queue held information including: an API key, List ID, and username (in the demo case only a username), a validation flag (for endpoint users making applications), and the image(s) to be processed (up to two - "front" and "back"). We queried this queue on a timed basis using a Quartz scheduled job. The quartz job runs in the background on a server or multiple servers and pulls from job queues to create Mechanical Turk hits based on the meta data. When a HIT is created the message on the job queue is moved to a processing queue. Once a Turk HIT is completed it is stored on Mechanical Turk for another quartz job to maintain. The other quartz job pulls from the Mechanical Turk storage and references the processing queue to see which HIT it was. When it realizes this HIT is completed it sends the XML output from Mechanical Turk to a Contact Service, and deletes the HIT from Turk.

Contact Service

The Contact Service is a groovy service we created that parses XML and transforms data into a Full Contact contact model that can be used by their many services. The contact service features any normalization needed to construct a contact model. Once completed, the service stores the contact model in the cloud (on s3) as a contact model data file.

Mechanical Turk

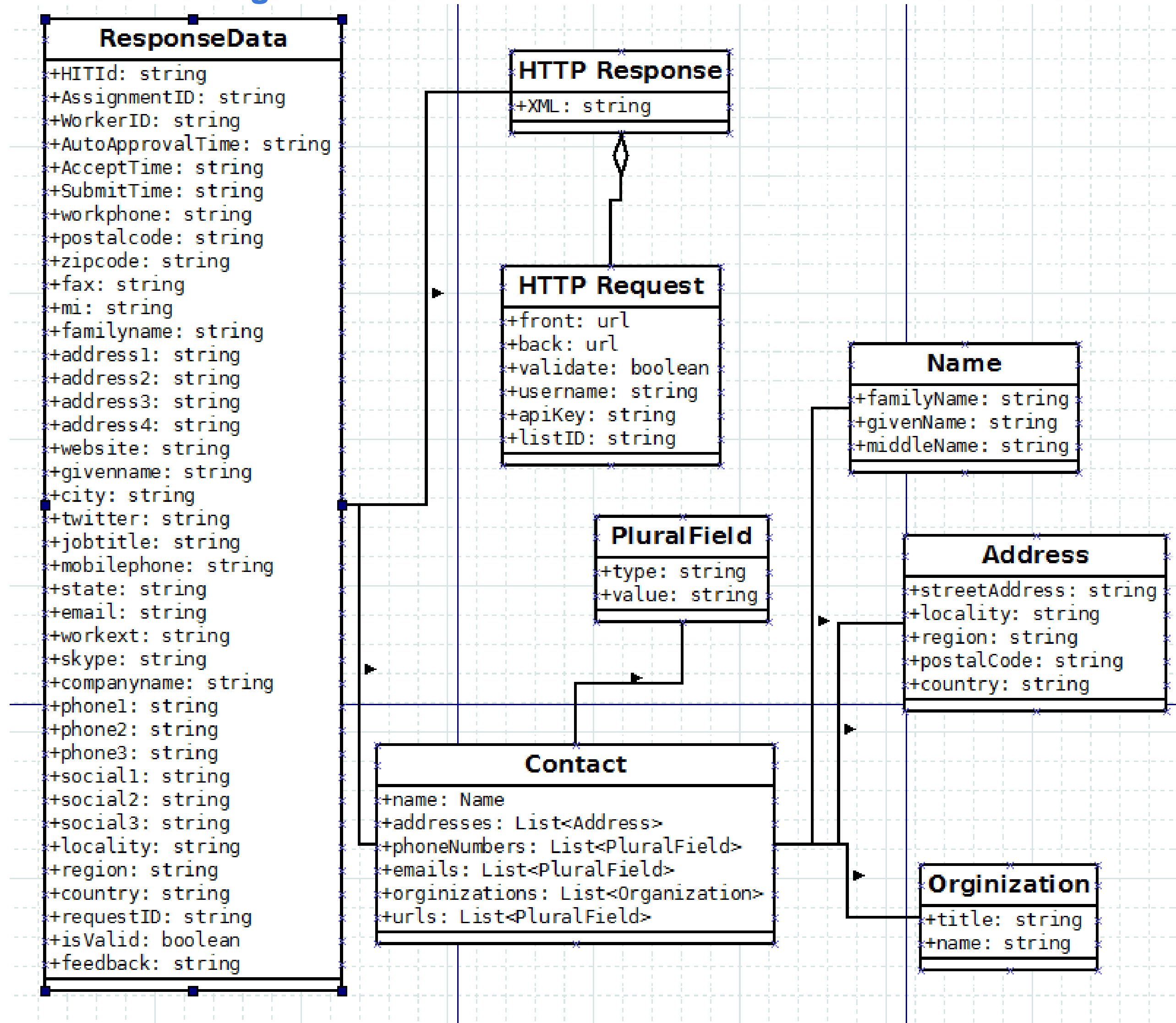
Mechanical Turk is an Amazon solution for outsourcing simple intelligence tasks to humans for a very small cost. Each task is identified as a Human Intelligence Task or HIT. Mechanical Turk was chosen because it ensures human level accuracy, fast turnaround time (sometimes less than a minute), and returns highly programmable data (XML output via HTTP response requiring no API resources for Mechanical Turk). Mechanical Turk also allows for automation of creating, managing, and reviewing HITs.

iFrame Solution

Mechanical Turk provided effective transcriptions because within the HTTP request made to create HITS we had the option to use an external iFrame (website) solution for Mechanical Turk workers to interact with. The goal of the iFrame was to have a highly usable, well designed web form that can send us data from the business cards and images via fields that are completed by workers. The iFrame uses jQuery as its main control scripting language. The motivation behind this was its cross browser compatibility and reliability. Since workers are not guaranteed to use any single browser, or have reliable internet, the iFrame was designed to reflect any technical limitations workers may have. The process of hitting the form with an HTTP request to completion followed a single flow: the page was sent images and an assignment ID

relevant to the Mechanical Turk HIT, the data fields on the form are completed by a worker, and all of this information including timestamps and worker ID's are sent to Mechanical Turk. This information is then pulled from an HTTP request that returns XML including all the metadata and completed image data from the HIT. Functionality in the form of image rotation and zooming was added to aid usability of the web form. Tab order was also added for fast keyboard navigation, while instant format checking for emails and URLs (red means bad, green means good) guided workers towards more normalized inputs. Overall the focus of the iFrame revolved around usability for Mechanical Turk workers, as well as accurate results.

Technical Design



Classes

The image above depicts the various main classes we used. Because of company confidentiality we chose to leave functionality out but the fields are all present and it is fairly obvious how they all tie together. The rest has been explained by the process in which it's gathered. However, the iFrame does most of this communication and is described in detail below.

iFrame

The iFrame was one of the biggest parts of the project. The simplicity of its look tells nothing about the complexity of the control scripts running it.

The web form is a modular, stand alone tool, that the application communicates with for every HIT. The web form accepts an HTTP request that contains information about which images to load, whether the assignment has an ID, and if it is double sided or not. Depending on the assignment ID (available or not) the web form produces a tutorial-like environment that does not allow submission, and uses an example business card with full functionality of the zoom and rotation for workers to play with before accepting a real HIT. When an assignment ID is available, a production HIT is created using the supplied images. These images show up as thumbnails on the left. One of the images is enlarged slightly and in most cases viewable from this state. However, every image has the ability for the user to mouse over and see a “zoom box” that zooms on the image for more clarity. The zoom box is a jQuery plugin. We made several changes to the plugin so that it would work with our implementation of the web form, and take into consideration our goal for extreme usability. Specifically, its use needed to not impede the progress of a worker or cause any frustration. The zoom was made to disappear at certain thresholds outside of the image so that entering fields or moving the mouse by the image didn’t cause user frustration. The zoom box size was also reduced and set to zoom to a certain level (so it is not too large or too small).

The images also have the ability to rotate. These rotations were trickier than anticipated because the images are not square. With this, the div containers that hold the images needed to be dynamically updated and refreshed with jQuery in response to image rotation. This also had to work smoothly and accurately cross browser. A lot of the spacing and containers rely on one another to keep everything in a specific position relative to one another so this was important to maintain stability.

The form fields also had a few tweaks to help the process. The email and website fields are dynamically checked on a timeout key press to ensure format is correct without bugging or confusing the user with too instant or too slow of feedback. The fields change color depending on the correctness of the format of the entered data. A drawback of this is that we used a set list of popular domains worldwide as a part of the regular expression. This helps from getting incorrect domains entered, but some random inconspicuous domains are not in the regex that could possibly turn the field red. We decided that this was a better idea than only using a certain character limit or size amount after the “.” in “.com” etc.. because a lot of URL’s have many different sizes (.museum for example) which are fairly popular. This would allow users to enter incorrect data and see green fields instantly where there is the possibility that they assume they are correct simply because of the green field.

The web form also uses Google Analytics to keep track of user types and habits. This will help improve the web form and improve usability. A few things we hope to track are users web browser types and page loading times. With the response from Mechanical Turk we can also tell how much time they spend on the page. These are some of the most important aspects to keep track of in terms of planning to decrease user frustration. Some future upgrades to the web form can include waiting to load images if page load time comes back slow, or seeing if certain browsers are showing slower completion times. This could mean that jQuery or other features are bugging out in different browsers and not showing up in tests.

Feedback is another feature the web form provides. Every hit has a feedback data field that is optional for Mechanical Turk users to complete. This is a small pop up box in the bottom left that allows users to tell us about the HIT they have just completed, or are currently working on. This data comes back with the HTTP response and can be emailed or stored for future review.

The web form also features potential of pulling a lot of information from an image. When first seeing the web form a set amount of fields are available. The form has the ability to dynamically add fields for certain groups of data such as phone numbers and address lines that will be sent to Turk. The application also has the ability to hand these and update the resulting contact model accordingly. The goal of designing these fields was to make it so it was as obvious as possible on how to add fields. Along with this design however, there are also hover-over tooltips that help notify users that "Add Line" does exactly that. These tooltips also guide users in case they have questions about how a field should look.

Quartz Jobs

We used a grails plugin called Quartz to regularly check for new messages and results from Mechanical Turk. Quartz allows the developer to schedule jobs using different triggers. We mainly used simple triggers to run our tasks. Simple triggers allowed us to give jobs names and specify different intervals to run different tasks. We created two quartz jobs for our project.

One job we created allowed us to check Amazon's SQS queue for new messages. Once a message is available, we take the string that we receive and parse out the necessary information. Once we have the information needed, the job creates a Mechanical Turk HIT using the image names. This job then removes the message from the preliminary queue, adds the HIT id to the string and pushes the new string a different "processing" queue. We used a simple trigger that scheduled this job to repeat every five seconds.

The other job that we created has a start time delay of five seconds. This job checks to see if there are new submission from mechanical turk workers. Once there is a new submission, it takes all of the post data and creates a contact model using the data. Once it has this model, it puts the text of the model into a text file, puts it onto the Amazon S3 system in a "results" bucket, then approves and deletes the HIT. This job uses a simple trigger and is repeated every five seconds.

Design and Implementation Decisions

The initial idea for the project was to use OCR (optical character recognition) software. We evaluated many different OCR SDKs. After testing many open source solutions and some of the top paid softwares that claimed to solve the problem we were faced with, we decided that the accuracy was not high enough to further pursue this solution. Instead we decided to use the online human service which is Amazon's Mechanical TURK. We chose this because, with the addition of a straightforward web form that is used when a TURK views the HIT, the accuracy goes up significantly.

One complication of this approach is knowing when a HIT is completed. We considered using web hooks to run a function whenever a HIT was submitted. Using this method was considered throughout the project. The reason for not using this method was the uncertainty of how to create a web hook, how to test the web hook, and the fact that another solution presented itself early on in the decision making process. We decided to go with a scheduling

plugin for grails called Quartz. Using this, we were able to use HTTP requests to check if there were submission available for review at an interval we define.

When a user submits a business card for transcription, the card must be stored somewhere, along with the user's information, including the username, list id, and access token. For the image file(s) we decided to use Amazon's S3 (Simple Storage Service). We decided to use this because this service is reliable, secure, and highly scalable. With the possibility of thousands of users submitting card images within a short period of time, the decision of which storage system to use focused highly on scalability and speed.

When an image is submitted, a message that indicates a new image has been submitted is created and stored. This message holds all of the information required for the system to create a HIT and retain all of the information needed to notify the correct user of the software. We wanted to stick to the centralized design that we used for storage of the images, so we decided to go with Amazon's Simple Queue Service (SQS). This allowed us to create a delimited string that contained all of the information we needed, push it onto this queue service, and pull it off when we needed to use it. This solution provided scalability and security all within a centralized service that can be accessed by any computer that has access to the internet.

Results

The API solution complies with the project descriptions and satisfies all requirements set forth by the formal requirements. Initial usability, accuracy, and precision tests have all yielded positive results. Usability is an ongoing iterative process with many changes aimed at Mechanical Turk workers. In terms of accuracy and precision, initial tests have shown the solution to return highly accurate results in a reasonable time frame. F-Score tests show a precision rate of .98 and a Recall rate of 1.00. Turnaround times for each request has been shown to be consistently less than 10 minutes. Looking forward, there are a number of potential improvements and additions that will increase the effectiveness of this solution.

One potential improvement is to use a database to store "scores" for different Mechanical TURK workers, which would allow us to give the worker of a business card a score based on whether the user accepted or rejected the result. This score could then be used to either block certain workers from future HITs if their work was rejected too many times, or give an extra incentive to workers that produce exceptional results. The implementation of a worker specific quality rating improves the likelihood of highly accurate results.

Another potential improvement is to revisit OCR technology. This could be used to prefill the fields in the Mechanical TURK iFrame, and instead of having the workers fill them all in, simply use them to correct the data that was produced by running the image through the OCR software. OCR technology also has the potential of delivering results instantaneously. Previous research has shown that character recognition could deliver near 100% accurate results. In combination with Mechanical Turk, the API would be able to complete a percentage of requests almost instantly which would result in lower turnaround times.

The iFrame in specific will undergo several more usability tests during its iterative process. Previous usability tests have resulted in changes that improved both accuracy and recall rates. Currently, the iFrame has an average use time of approximately 4 minutes per assignment which means a typical worker spends 4 minutes on the iFrame reading and transcribing the business card. Feedback and testing are currently being used to make small usability improvements on the iFrame aimed at increasing worker efficiency and accuracy. The goal of all testing will be to improve the iFrame to decrease the time workers spend on the iFrame and to increase accuracy. All tests are conducted on a set of 48 business cards where results are compared to a truth set of the same 48 business cards.

Appendices

Acceptance Tests

So far the company employees have gone through some acceptance tests to decide on many of the changes that were made to the web form. Further testing is planned to be done with actual Mechanical Turk workers through mass testing (hundreds of users and images) seeing what feedback they have about the web form process.