Team: FullContact #2

Client: FullContact

Team Members:

John Galbavy

Robert Keith Rippetoe

June 24, 2016

Introduction

FullContact is the most powerful, fully-connected contact management platform for professionals and enterprises that need to master their contacts and be awesome with people. FullContact's cross-platform suite of Apps and APIs enhance contacts with 360-degree insights, while keeping them organized, synchronized, up-to-date, and safe. FullContact first started in 2010 with 100 beta contacts, later developing into a startup focused on synchronizing contact information into a succinct and organized platform.

Social media is an omnipresent and inescapable part of modern living. From Facebook to Linkedin, social media companies offer services to turn social relationships into easily transferrable data. One service of social media companies is that of "tagging", attaching a small portion of information to a contact in order to easily categorize or identify the tagged contact. Because our client, FullContact, is an all-in-one address book by connecting multiple accounts into one place, they would like to implement this tagging feature. This feature follows the FullContact policy of being easy to use and requiring as little effort as possible for the user, hence the smart aspect. Unique to FullContact, these "smart-tags" can draw upon information from a wide variety of different data sources, depending on what the user has connected. These could be twitter feeds, email signatures, or even public personal information. Using the different data FullContact has acquired, we created Smart Tag suggestions that offer desired information to even the most novice user in a visually appealing and intuitive way. By developing this project, FullContact will present tagging services to the users that will offer Smart Tag suggestions which the user can then update their contacts with a simple click.

Requirements

High-Level Description/Vision

Smart tagging will be used as a part of Full Contact's social media platform application. The product will be targeted based on the existing customers of the Full Contact user base, mainly outward facing business professionals. Smart tagging will address the issue of categorizing large amounts of user information into easily searchable categories without requiring effort from the user. Smart tagging must accurately create and suggest Tags based on Contact Information.

Functional

Given an address book, produce a list of suggestions for each contact within the address book with suggestions being a list of smart tags unique to the user. This will be done by sending the address book through a series of smart tag modules, each module representing a different tag. Smart tags can then be created using Machine Learning libraries where the normalized contact information can be extrapolated upon to produce smart tags. If a tag is created, a suggestion can then be made for that contact. Each module will have unique algorithms for creating a smart tag

based on data within the tag and data available through other means (API's, contact of contacts data, etc.).

Non-Functional

- The project must be scalable to real time data.
- The project must provide tags that are beyond obvious data and direct connections.
- Code must be able to be integrated to FullContact API.
- Code must massage data to a useable standard.
- Project must be built from existing FullContact infrastructure.

Technological Risks

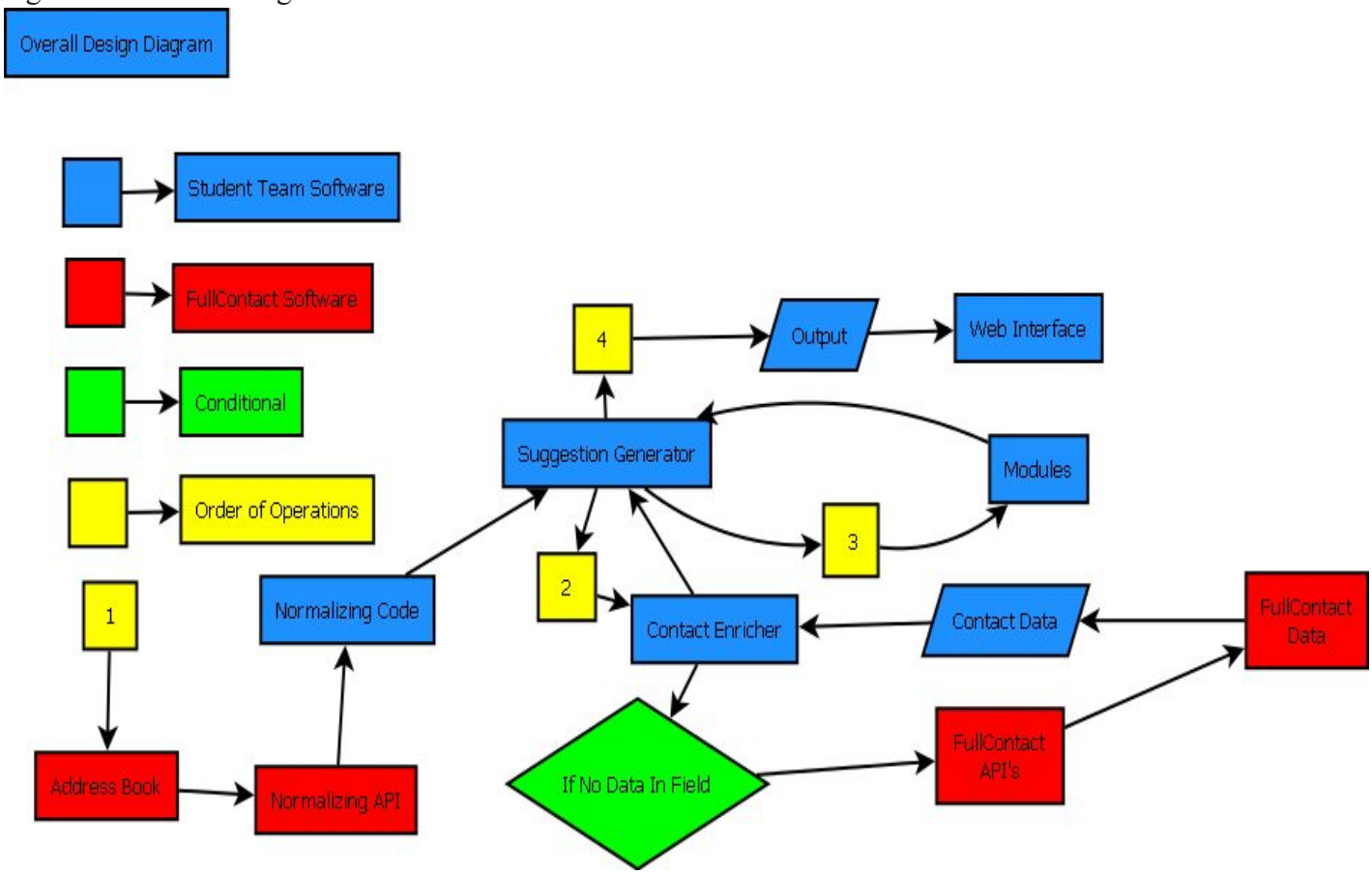
- Dissemination of private information to the public
- Project can be tested to be functional, but when integrated does not perform sufficiently on real time data.
- Risk of being too processor intensive for real time data, necessitating a more optimal base algorithm.
- Smart Tags developed in our time frame may not be diverse to be released on the main application platform.

System Architecture

The system architecture of our project (Figure 1) outlines the overall flow and interactions of the project. There are four key sections of code, with Suggestion Generator being the central component controlling the flow of contacts. In order to maintain the scalability of the project to real world data, Suggestion Generator is responsible sending one contact at a time to either enrich the contact or to the submodules and then to aggregate all the smart tag suggestions made by the submodules into one. The four key sections are as follows:

1. The input to our program is an address book. This input then has to be normalized using a FullContact api and then the updated information extracted into a list of contacts, which can then be sent to the Suggestion Generator to start the smart tag create process.
2. After Suggestion Generator receives the normalized list of contacts, it will then delegate to our Contact Enricher one contact at a time to check for more data on the contact, which will then return an enriched contact list to the smart tag generator.
3. With the list of contacts now containing as much information as available, they then can be sent one a time to the submodules. Each submodule is responsible for creating and returning a smart tag suggestion for its given tag type. That is to say, the gender module is responsible for creating both male and female smart tags and returning the suggestions for both smart tags.
4. Suggestion Generator can then combine every submodules' suggestions into one list of suggestions. This completed list of suggestions can then be sent to our output, which can then be viewed on our web interface.

Figure 1. Overall Design



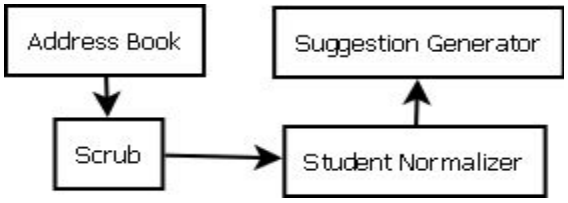
Technical Design

Normalization Design

To begin the program, an address book is inputted, and then must be normalized, which will create a list of normalized contacts (Figure 2). These contact objects follow the FullContact schema of data and pull their data right out of an address book. This list of contacts is then sent through a normalizing API from FullContact called scrub which normalizes the contact's data fields by removing special characters and mapping data entries to a common form. An example of this would given a contact, Contact A, whose job title is "C.E.O" and another contact, Contact B, whose job title is "Chief Executive Officer" the scrub API will then return the common form of "CEO" for both of these. The Scrub normalizer does not fully reform the data, so we implemented an additional normalizer specific to each individual module. This addition mostly reforms the data fields into more manageable forms, like splitting the job title of "CEO / Co-Founder" into two titles of "CEO" and "Co-Founder". Normalization plays a key role in the generation of smart tags since it attempts to reduce all data entries to a common form with only a few exceptions. The normalized data fields can then replace or update the inputted contact list,

which can then be sent to Suggestion Generator to find more information if needed or to start creating tags.

Figure 2 - Normalization



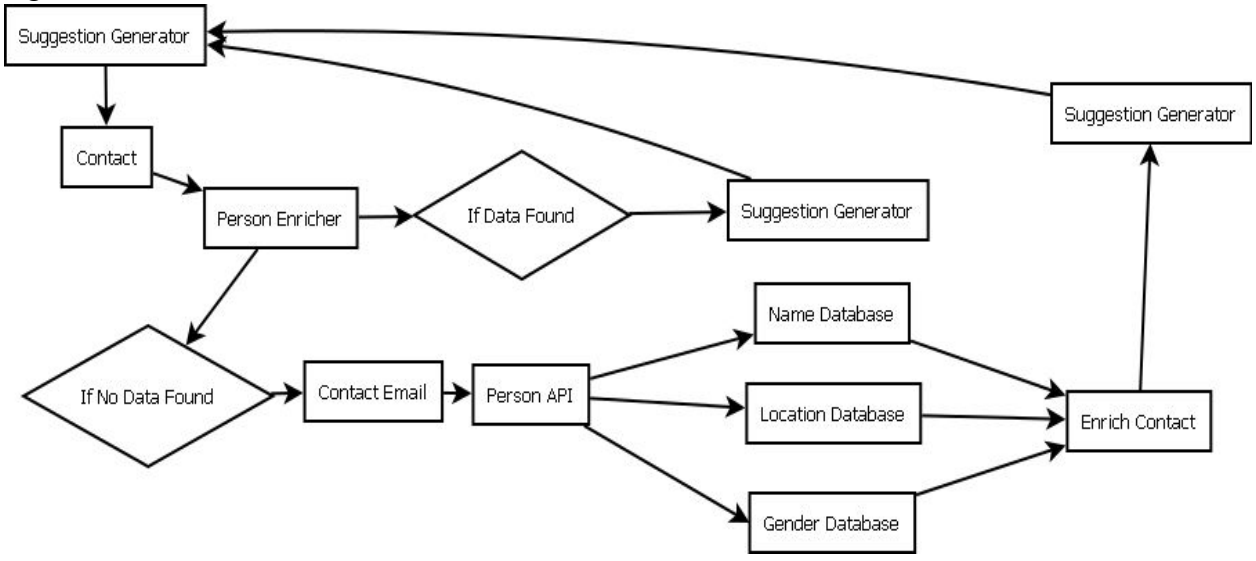
Contact Enrichment

Now that each contact's data is normalized, we then want to get as much information as possible on the contacts to either update current data or to fill in missing data. This is the Enrich Contact process in Figure 1. This process then connects to two different API's, Person and Company, which try to update different data. Person API looks to update data fields corresponding to personal information like name, gender, and location. Company API looks to update company name and job title. Each of these data fields are what the submodules will be looking for when they create smart tag suggestions. Both Person API and Company API will now be expanded on in more detail.

Contact Enrichment - Person API

Suggestion Generator will pass off one contact at a time to the Person Enricher to check for possible information updates (Figure 3). If information is present, the Suggestion Generator starts the process again with another contact since there is no need to update the contact. If information is not present, the Person Enricher searches for an email to do a PersonAPI lookup to find more information. When an email is found, the Person API supplies the Name, Location, and Gender of the contact if possible. This is used to fill empty contact fields before the Suggestion Generator begins the process again until all contacts have been processed.

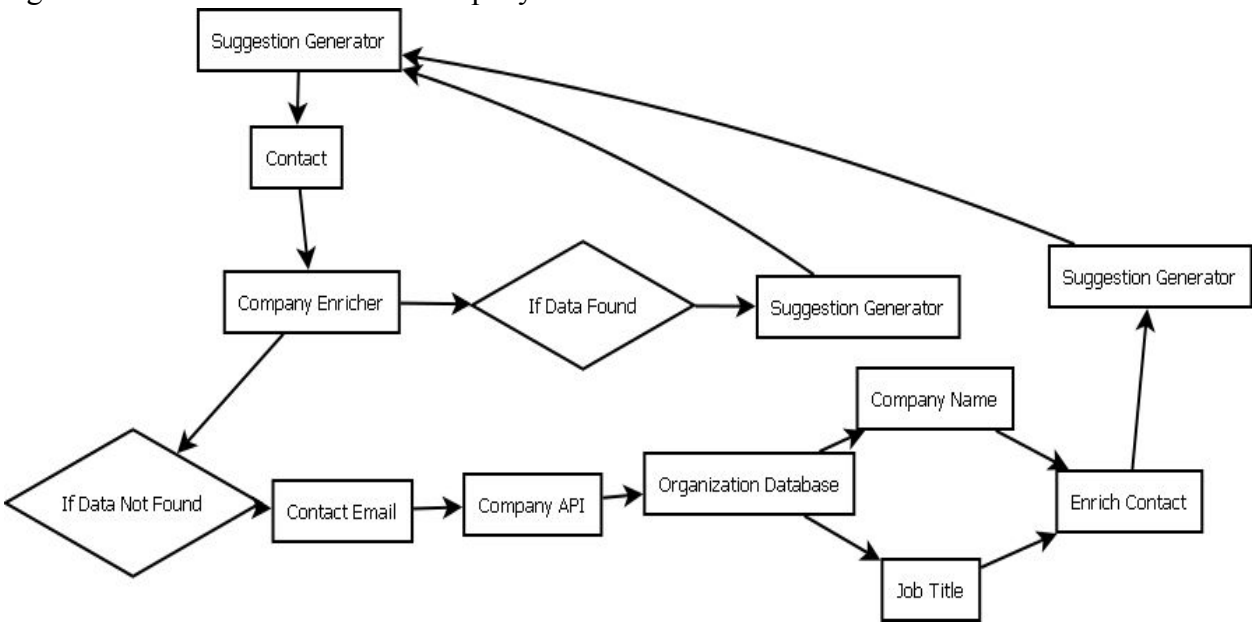
Figure 3 - Person API



Contact Enrichment - Company API

The company enrichment process follows the same design as the person enrichment process, except for trying to update different fields (Figure 4). If data is not found for a contact in the job title or company name fields, this service will then attempt to look up the data in the organization data base. If the organization database has updated information on the contact, the service will then enrich the contact and return the updated contact to the Suggestion Generator.

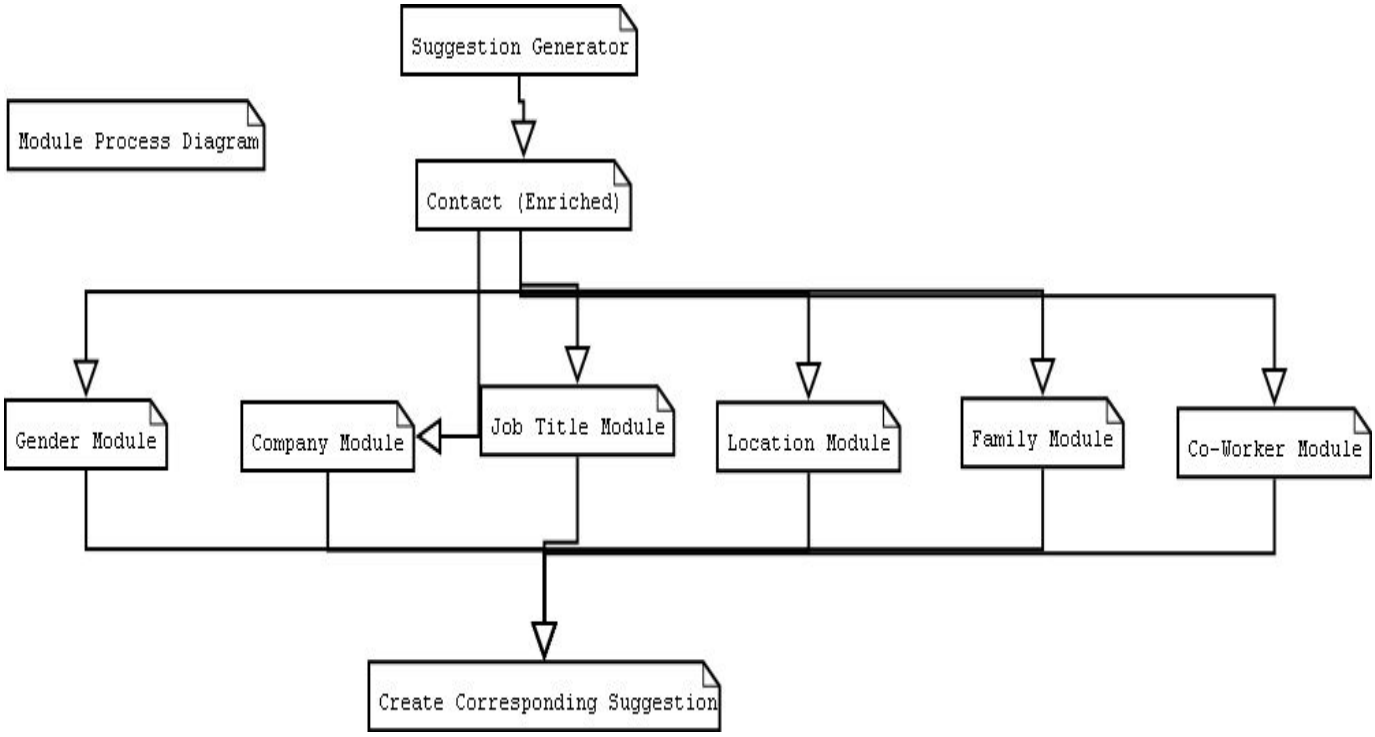
Figure 4- Contact Enrichment - Company API



Submodules Design

To understand the individual submodules created to parse through contact data, it is important to recognize the structure of the FullContact address book. Essentially, each contact has four main fields: name, location, gender, and organization. Other contact information is mostly FullContact metadata. From the main fields, six different modules parse through this data as well as making direct connections relative to the owner through family and coworkers (Figure 5). Each submodule is responsible for creating a certain tag type which corresponds to different contact information fields. An example of this would be the Company module is responsible for checking the contact's organization names and creating smart tags relative to the company names. When smart tags are created, the module is then responsible for creating a Suggestion, which is really just a map of smart tags to the contacts they are tagging. The modules then return the suggestion(s) they created to the Suggestion Generator to be aggregated.

Figure 5 - Submodules Design



Design Decisions

Our first design decision was how we were going to input contacts and create Smart Tag suggestions for those contacts. We eventually decided on inputting a list of contacts representing a person's address book. This is testable for our team since we can create a JSON file of contacts and easily run it while remaining scalable to a database set up. After we take in our input of contacts, we then chose to normalize the list using FullContact Scrub APIi. Even though the API has little documentation and is not straightforward to use, it was created specifically for contact objects found in FullContact address books, so we had no need to modify it for our uses. Because of this decision, we implemented our own addition to the Scrub normalization for each module as discussed in Technical Details.

We decided to give Smart Tags a specific tag type which relates to the contact data we can access. For example, labelling a tag "Company" which corresponds to the contact's data field of "Organization" gives us an easy way of categorizing them. Each Smart Tag was then also given a UUID making it unique. This gives Smart Tag s scalability for a database implementation. Lastly, Smart Tag s were also given a name field which will eventually be displayed to the user for what the Smart Tag represents ("Male", "Denver", etc). Then a Suggestion can be created for each Smart Tag . These suggestions are a map of Smart Tags to a

list of Contacts that have been tagged to it. We initially had suggestions containing the Smart Tag UUID but changed this to the Smart Tag in order to give us more information to work with for comparisons and other relations.

We chose to set up the skeleton of generating Smart Tag suggestions by creating a service that delegates the list of contacts to each submodule one contact at a time referred to as Suggestion Generator in figure 1. We chose this setup to preserve the scalability of the program and allow for our submodules to individually create their suggestions that Suggestion Generator will then be adding to the overall list of suggestions when they have completed. Each submodule was created to relate to a specific Smart Tag's tag-type. This allows more submodules to be easily added to the underlying skeleton since each submodule is in charge of creating its own suggestions as well as keeping a categorical structure for Smart Tags.

In order to create more Smart Tags, we expanded our program to implement a machine learning library. This is to produce Smart Tags that go beyond the basic contact information provided. For two reasons, we chose the machine learning library Weka. First, it has the largest library of diverse machine learning algorithms with detailed documentation. This is very helpful since we did not know which algorithm would solve our Smart Tag problem, so we needed to be able to test many different algorithms and compare how well they solved complicated tagging issues. Second, Weka has a GUI that is simple and easy to use. A simple GUI is important to test machine algorithms easily without investing much time into programming a failing algorithm.

About a week of time was initially spent attempting to utilize a machine learning library called Weka to analyze contacts. To do this, contact data was converted into ARFF files designed specifically for Weka. Although the data was initially successfully converted into ARFF files, the algorithms provided by the Weka library were too restricted to be integrated into our data and were ultimately not part of our final project. Initially, the algorithm's main purpose was to use machine learning to draw connections beyond what our modules were capable of. This failed both because of the limited amount of information available and the the degree to which non-machine normalization already succeeded in removing defects, leaving little to improve upon. In the future, these Weka algorithms could be used and the ARFF code reintegrated if they are deemed useful to the project.

Results

Our program was developed with scaling as a priority, but there currently are no quantitative measures for how it will perform as an API. The project was presented at a final representation before the FullContact organization, with which it was welcomed as a presentable and well developed demo. Both Keith and John will be hired to work further on the

smart-tagging project, developing the project as an API that can be applied to an address book as well as being integrated into the FullContact database and the beta of the future FullContact web application. One of the lessons we learned throughout this project was that deprecated code is not entirely to be avoided, demonstrated through our use of deprecated code as the only way of implementing our normalizer. We learned that API code many times has much less information than would be useful for a complete novice, and overcame this by seeking another individual with more complete knowledge. Although at the time that this report is being submitted the project has yet to be integrated into FullContact software, both members that worked on the smart-tagging project were offered internships in hopes of continued work on the project and eventual integration with the FullContact web application.

The list of features that have still yet to be committed because of time constraints are vast, ranging from machine learning to data reconfiguration. A If converted into an API, the smart tagging program could be used in the same capacity as a web application. Furthermore, now that the suggestions are clearly visible, further normalizations beyond the Scrub API (normalizing software provided by FullContact) can be developed.