# CSCI 370, Summer 2012
# FullContact
# NLP / Information Extraction

Ryan Lewellen, Jason Lingle, Steven Ruppert

June 2012

## 1 Introduction

FullContact is a Denver-based startup specialising in extraction and aggregation of personal contact information. Their flagship product provides users with an address book system which automatically enriches, deduplicates, and updates entries originally added by each user.

FullContact has commissioned our team to develop a system which extracts personal contact information from user-supplied Internet mail messages. For example, given a MIME-format message with the content

```
...
From: Alice Adams <aadams@acme.com>
To: Bob Brown <bob@example.org>
...
Hi Bob,
Attached is the invoice of the product you ordered.

Alice Adams
Sales Manager, ACME, Inc.
+1 800 123 4567
```

we should ideally extract the following information:

```
Given name: Alice
Family name: Adams
Email address: aadams@acme.com
Position: Sales Manager
Company: ACME, Inc
Phone number: +1 800 123 4567
```

# 2 Requirements

## 2.1 Functional

The software system must take a MIME-format mail message, as a string or a Mime4J object, process it, and return a Contact object containing what information could be extracted.

If possible in the allotted time, the software should present itself as a REST API.

## 2.2 Non-Functional

The software system must be written in a language that runs on the Java Virtual Machine, preferably Groovy, Clojure, or Java, in that order of preference. Other JVM-based languages, such as Scala, are permissible but discouraged. The build process and dependencies must be managed by Gradle. The main part of the software must make no assumptions about the underlying operating system or environment.

The data extraction process for an average message must take fewer than 250 milliseconds of real time on a typical computing system. This performance requirement excludes any pre-process initialisation which occurs one time, before or when the first message is processed.

The types of data which should be extracted are as follows, in order of decreasing priority: Names (given and family), email addresses, phone numbers, physical addresses, social media data (such as Twitter handles, LinkedIn profiles, etc.) companies, positions, honourific prefixes and suffixes.

Data extraction should be optimised for precision rather than recall. That is, it should favour the *correctness* of its returned data over the *quantity* of the data; it is preferable that the user need to manually enter data than have to worry about whether the extracted data could be wrong.

Linguistic structure permitting, the data extraction system must function for non-English locales as well as English. This includes proper handling of non-ASCII characters, accounting for character encodings and email clients which have broken encoders, and support of non-American systems, such as phone number and physical address formats.

In order to facilitate scalability, the software should be parallelisable, at least able to run independent jobs simultaneously.

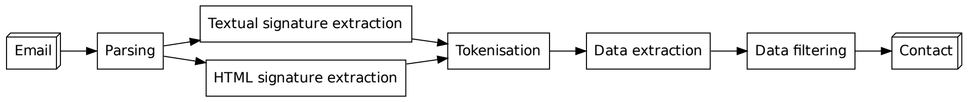

Figure 1: Overall architecture of the software system.

## 3 System Architecture

Due to the narrow purpose of this project, the higher-level architecture is quite simple. Reference figure 1 for a process diagramme.

Input to the system is provided in the form of a MIME message, either as a string or as an Apache Mime4j `Message` object. In the case of a string, the message is converted to a `Message` object implicitly.

Once a message has been obtained, the parser will then extract the text and html bodies (if present). The HTML section, if present, will be sanitised using HTML cleaner. If no text section is present, then it will be constructed from the HTML section by stripping any and all tags.

Any hint-able contact information is also extracted from the MIME message headers; this information includes the person's name (if specified) as well as any email addresses that are specified in the `From` header as well as the `Reply-To` header.

Given the text/html bodies of the message, the system will then attempt to determine which portion of each body is considered the signature of the message, so that the data extractor does not have to sift through the whole message to get at the contact information.

The extracted signature is passed on to the tokenisation system. The tokeniser not only separates word-like pieces from each other, but also attempts to deduce a certain tree-like structure from its input. In the case of text, this structure is defined by the hard line breaks contained; for HTML, it is simply the hierarchy of nodes.

The resulting tokens are then passed to the data extraction system, which is responsible for searching for contact data within the signature. The data extraction system is divided into a group of data extractor sets. Each data extractor set is responsible for setting the entry for extracted contact data and for reconciling multiple results. A data extractor set consists of multiple data extractors, each of which can extract data of the appropriate type and

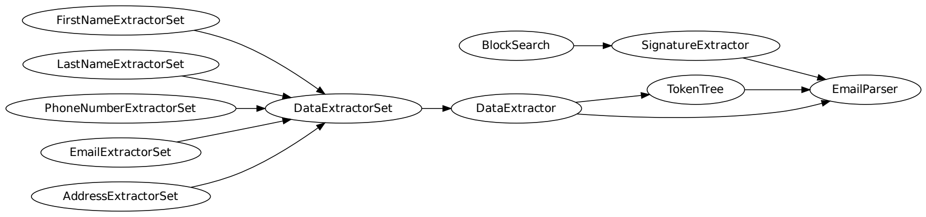

Figure 2: Upper-level classes of the Software System and the primary relations between them.

estimate the certainty of the correctness of such results.

After data extraction completes, the software takes a second holistic look at the extracted contact data to check for certain types of errors. For example, in certain poorly-formated documents, it is possible to pick up the same word for both given and family names; this indicates that the contact's name is not reliably known and that the name should be removed.

## 4 Technical Design

Our software is, as a whole, laid out in a tree structure, each part performing some work and delegating the rest to subordinate objects, with the number of cross edges minimised. Figure 2 shows the hierarchy of responsibility, and figure 3 illustrates the flow of data between the components.

The root of the API is the `EmailParser` class, which handles decoding/cleaning of email messages, as well as invoking the signature extractor, tokeniser, data extractor, and post-extraction filters.

The `SignatureExtractor` (implemented by `BlockSearch`) has the sole responsibility of identifying the signature in a given mail message, when possible. It handles both text and HTML signature extraction, since the two are necessarily similar and partially interdependent.

`TokenTree` is used to convert disparate input types, such as plaintext, HTML, or a DOM node, into a uniform data structure used by the data extraction system.

`DataExtractor` itself does very little. It simply contains a number of

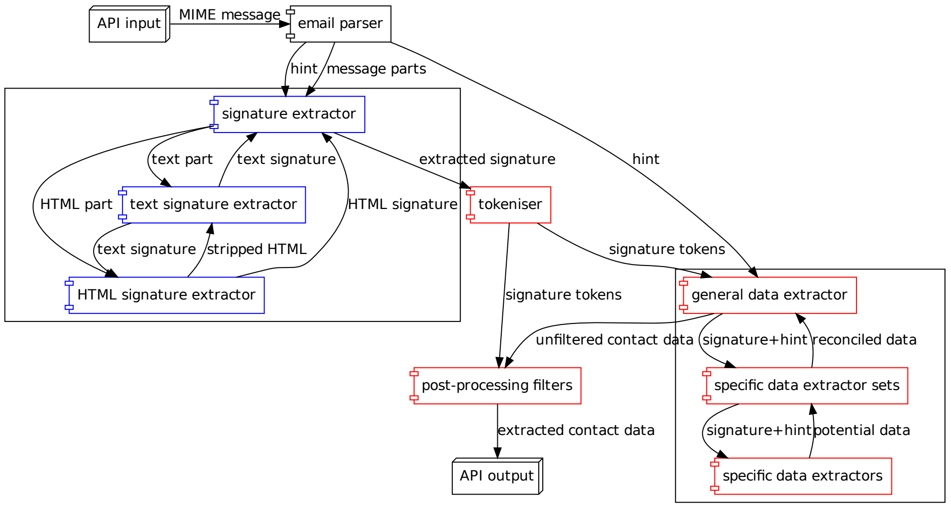

Figure 3: Components of the Software System and the flow of data between them. Items have been consolidated and/or separated to increase clarity. Multiple edges used for simple data forwarding have been joined into cross edges.

`DataExtractorSets`, each of which it calls in sequence to extract each data set, and to perform post-extraction filtering.

Each `DataExtractorSet` subclass, like `DataExtractor`, functions primarily as a container delegating most of its work to the contained `Extractors` (not shown in the figures above). It also defines the logic to write the extracted data into a Contact object, as well as how to reconcile multiple disagreeing extracted data.

The `Extractor` subclasses are the lowest-level in the hierarchy. Each defines a method of extracting data from a signature or from provided hints and to determine the certainty that each datum is correct.

## 5 Design & Implementation Decisions

Our first design decision was the primary language in which to implement the system. Confined to the Java Virtual Machine, we chose the Groovy language, because it remained close to the traditional Java syntax, while being considerably less verbose and supporting additional features like closures, array literals, and run-time dynamic metaprogramming. Other JVM languages such as Scala, JRuby, and Clojure were considered but were not used because using Groovy enabled us to build on developer knowledge from the rest of FullContact, who use Groovy extensively.

Our system was compiled and built by Gradle, a Groovy-based build system. Gradle has an ease-of-use advantage over similar Java build systems like Maven and Ant, while still being accessible from the command line as opposed to a build system integrated in an IDE such as Eclipse. However, the long startup times of Gradle proved an annoyance during development.

When researching methods of extracting contact information from emails, we found several research papers on the subject, detailing the methods used. A paper from Microsoft Research described a machine-learning approach to extracting the signature blocks from emails, using a Java-based machine learning library (MinorThird). We opted instead for a heuristic-based approach, due to the time constraints of the project and the time required to deliver partially-finished but working software during Scrum sprints.

This method involved first attempting to find what we consider the signature in the message, and then running a second step to extract all of the contact information from this extracted signature. This made it easy to separate concerns of making sure that we have the correct contact information and finding the information in the unstructured docuent.

The scheme for signature extraction involves first segmenting the text into what we refer to as "blocks" - these are segents of text that are separated by non-content lines, which consist of empty lines or lines containing only a single, repeated character. This simple heuristic does a very good job of separating a person's signature from other parts of the email, such as the actual body of the message.

To choose which of these blocks represented the signature, we assigned each block a score based on the presence of certain "features"; we gave positive scores for things like the users' first name, last name, and email address, and negative scores for things like the presence of the recipient's name/email, presence of MIME headers, and indicators of reply chains.

This works fairly decently, but sometimes people do weird things with their signatures, such as double spacing their signature, or putting their signature into two or more adjacent blocks. By adding some block refinement heuristics, we were able to deal with these issues.

Data field extraction (emails, names, etc) were also handled with a heuristic approach. While an AI-based machine-learning approach might have higher accuracy in the long run, we again opted for a primarily regular expression-based approach to minimize iteration time. However, to enable the product to scale as the test data corpus and accuracy requirements grow, we designed both the signature extraction and data extraction interfaces to facilitate switching to an AI approach when a sufficient training corpus can be built.

Name extraction has by far the most extraction methods, and was also the most difficult to get to work precisely. Originally, we used only a Markov model to determine the most name-looking items; however, this had precision of around 30%. We since added ways to extract names based on common patterns within signatures, the display name provided in the MIME headers, as well as deducing it from common email address formats.

Email address and phone number extraction in many ways work similarly. Each has two primary extraction methods: One uses a simple pattern that will find most data, but with lower certainty; another uses a more comprehensive check against known common formats and yields much higher certainty. In particular, we compiled a list of common phone number patterns for nearly every major country. Email addresses may also be extracted from the MIME headers, a very reliable method which usually outperforms signature-based extraction.

Physical addresses are the hardest to extract (though not the hardest to make correct, which falls to names), due to the varying and non-specific formating and the great number of corner-cases. American-format addresses, also used in Canada and Singapore, are the easiest to work with, since they begin with a house number and end with a postal code, possibly followed by a country. European-format addresses (used in many parts of the world), which place the postal code before the city and may place the house number on either side of the street name, were significantly more difficult. Only a minority of such addresses are located successfully using our methods. We were never able to determine an effective method of identifying Indian-format addresses which did not involve obtaining a massive list of known city/municipality pairs.

Early in the project, we realized that being able to easily run the system against test data and get summary metrics on the performance of the system would be extremely valuable, both for developer feedback and as a visually accessible product of the programmatic API. We thus created a number of supporting Groovy classes to load structured, human-generated test data, run the original email against the system, and report the results of the extraction. Because this testing takes a fairly long time (about two minutes on a netbook with 1GB of RAM), we separated the script that runs our tests, so the short-running unit tests can be run separately from the accuracy or "integration" tests.

We decided to store our test data in a JSON format, with the original MIME email as a string in a "notes" field. This rather unintuitive structure is an artifact of our use of an existing API to convert the JSON into FullContact's internal Contact data structure.

However, we ran into problems using the `JsonSlurper` JSON-parsing class in Groovy's standard library to parse our test data into a native Groovy data structures. Because the `JsonSlurper` implementation uses Java JDK's Regular Expression implementation to validate JSON format, our test data triggered a bug where the alternation within the regular expression would lead to a `StackOverflowException` when JSON strings were greater than a certain width; the original MIME-formatted messages stored in the JSON often triggered this.

To work around this, we developed a more efficient and functionally equivalent class `JsonDiner` that was able to parse our test data into Groovy data structures without any problems. The class is written in Java for higher performance, and is checked into the version control with the rest of the project, along with its unit tests. Due to the annoying bug in `JsonSlurper`, as well as the other peculiarities of JSON's format, future revisions of the system should consider migrating to a more human-manageable format, such as YAML.

Rather than simple console output for our test suite, we spent some time creating an HTML report format, so we could use the additional formatting options of CSS to highlight elements such as correct vs incorrect results visually, while remaining easy to view (in a web browser).

## 6 Results

As an API, the system has no real UI, graphical or otherwise, and instead exposes its functionality programmatically, in the form of the structured Contact model and associated metrics with the extraction process. To facilitate quick developer feedback on how the system performs, the system is capable of generating detailed reports about the extraction process as compared to our corpus of human-extracted contacts from test emails. The test corpus is stored in JSON format, and the system can automatically be run against the corpus with the TestSuite tests. The generated reports are in HTML format, and contain the expected and actual contact information, as well as visual indication of how the signature extractor classified the parts of the email, and where data was extracted inside the actual signature.

The overall performance of the system in each of the extracted information categories is summarized in table 1 by the number of True Positive, True Negative, False Positive, and False Negative results in relation to the expected human-extracted contact information . These can further be summarized by the Recall, Precision, and F1 score (the harmonic mean of recall and precision). Our approach sought to optimize precision over recall, in that the system would only emit contact information with a high certainty of existing at the expense of missing some contact information that could have been extracted to avoid incorrect data (false positives). Running the final system against the test corpus produced the following results:

While the performance of the system on the currently extracted fields is excellent with the test corpus, future work should extend the performance to both a much larger corpus of emails, as well as other relevant fields, including organizations (Company, Title), and social media links (Twitter, Facebook, LinkedIn). The modular nature of the system will facilitate the integration

|  | Signature | Name | Address | Phone | Email |
|---|---|---|---|---|---|
| **True Positive** | 408 | 143 | 17 | 82 | 151 |
| **True Negative** | 6256 | 5 | 130 | 56 | 2 |
| **False Positive** | 126 | 3 | 0 | 0 | 1 |
| **False Negative** | 182 | 3 | 7 | 16 | 0 |
| **Precision** | 76.40% | 97.95% | 100.00% | 100.00% | 99.34% |
| **Recall** | 69.15% | 97.95% | 70.83% | 83.67% | 100.00% |
| $F_1$ **Score** | 72.60% | 97.95% | 82.93% | 91.11% | 99.67% |

Table 1: Overall results at the end of field session

of these additional fields and test data.