FullContact

Building a Living Social Network

Developing a Connections API – How well do you know your contacts?

Travis Boyd
Martin Kuchta
Daniel Romero
June 17, 2014

Table of Contents

Client Description

FullContact, a startup company based out of Denver, is solving the world's address book problem. Users in this day and age are constantly connecting to people, constantly finding new people, and constantly changing. The address book problem is the idea that all of these constant changes in contact information create this massive web of complications throughout each user's address books. The goal of FullContact is to create and maintain a single resource where any user can go to access all the information they want at any time, despite the ever-changing world of communication.

Whether the change that happens is that you move jobs, move away from people, or make new connections, FullContact aims to use its web and mobile applications, along with an advanced API, to keep up with that change. The company has been able to productize the idea of solving this problem, with great success.

Although users may seem to be the main focus for the company, most of the real work is done by organizing the contacts for companies. Handling and updating an address book is one thing, but managing and maintaining a company-wide log of communications between employees, board members, human resources staff, customers, clients, and so many more people is a problem best left to the experts. And FullContact believes that they are the experts.

So what is it they really *do*? Well, contacts in any address book are constantly becoming obsolete. People change their contact information, or they may gain a whole bunch of new information from new business cards or phone calls with new employers, friends, family, etc. In a regular address book, users must enter all this new data in by hand. Many don't, they simply leave the contact out or only fill in the necessary parts. But FullContact can take these partial contacts and create full contacts, by keeping track of data across accounts on many different services. If you have partial information about one of your contacts in five different address books, FullContact's Address Book can consolidate all of this information, intelligently resolving conflicts to get the most current and accurate information. The address book problem is quickly becoming a thing of the past.

Product Vision

When thinking about organizing an address book, one thing to consider is the idea of connection strength between two people. Could it be possible to tell who you're most strongly connected to purely based off of the level communication you've had with that person in, say, the last 30 days? The goal of our project was to find this connection strength. Based off of both past and present, sent and received, human or spam bot e-mails, the team developed a utility for the company that quantifies the relationship between users.

Since the potential amount of data that could be used in the future is enormous, developing a backend capable of scaling to that level was essential. Another goal of the project was to utilize different technologies to handle this significant amount of data through the use of an efficient data store to log communications, as well as a database to store the connection data. The team recognized also that analyzing this amount of data would be difficult and slow without considerable optimization of the analysis algorithm and upload/download time from servers.

It is important to recognize that e-mails are not the only form of communication today in the modern technological world. Twitter, Facebook, phone calls, text messaging threads, and many other forms of communication are essential to staying connected to other people in the world. Although it was not critical to implement all of these different forms of communication, it was important to generalize the build enough to allow for easy and quick implementation of any and all new forms of contact.

Requirements

I. High level description

FullContact is a networking company interested in accessing user contacts, connection data, and other useful information in order to fully connect people together in a digital landscape. By gathering data such as e-mails, address books, or business cards, the project we developed uses this data to infer relationships between users and their contacts. For the purposes of our project, a graph would be created and maintained by a remote server. Each node in the graph represents a contact, each with its own contact information like e-mail address or phone number, and each edge represents an event that occurs between two users, like an e-mail or a phone call.

Our overarching goal is to be able to, for a given time range, query the strength of a relationship between contacts from a log of past communication. This can be as simple as listing out a number showing communication level among users, or as complex as displaying to the user an intricate web of connections and relationships, each with their own strengths and methods of communication. Specifically, we should be able to query the database for a list of a specified number of top connections.

The objective is to establish a proof of concept for determining strengths of relationships, to be used later in production applications. We must also generalize the code base enough to allow for the quick addition of further forms of communication.

II. Functional requirements

- Allow extraction of communication data from multiple sources.
    - This can range from grabbing e-mail metadata to getting permission to read contacts, Twitter accounts, Facebook posts, LinkedIn profiles, etc.
- All forms of communication must be able to be represented by the same data structure.
    - Data should not be separated by type of communication.
- Communication data must be persisted in a log.
- Allow for retrieval of time slices of communication data from an API endpoint.
    - If other teams at FullContact want raw data to use for their own analysis, they should be able to easily obtain it.
- There should be a way to retrieve a list of top connections from an API endpoint.
- Periodically update stored top connections.
    - Performance for this is not critical, but there should be no production downtime while it happens.

III. Non-functional requirements

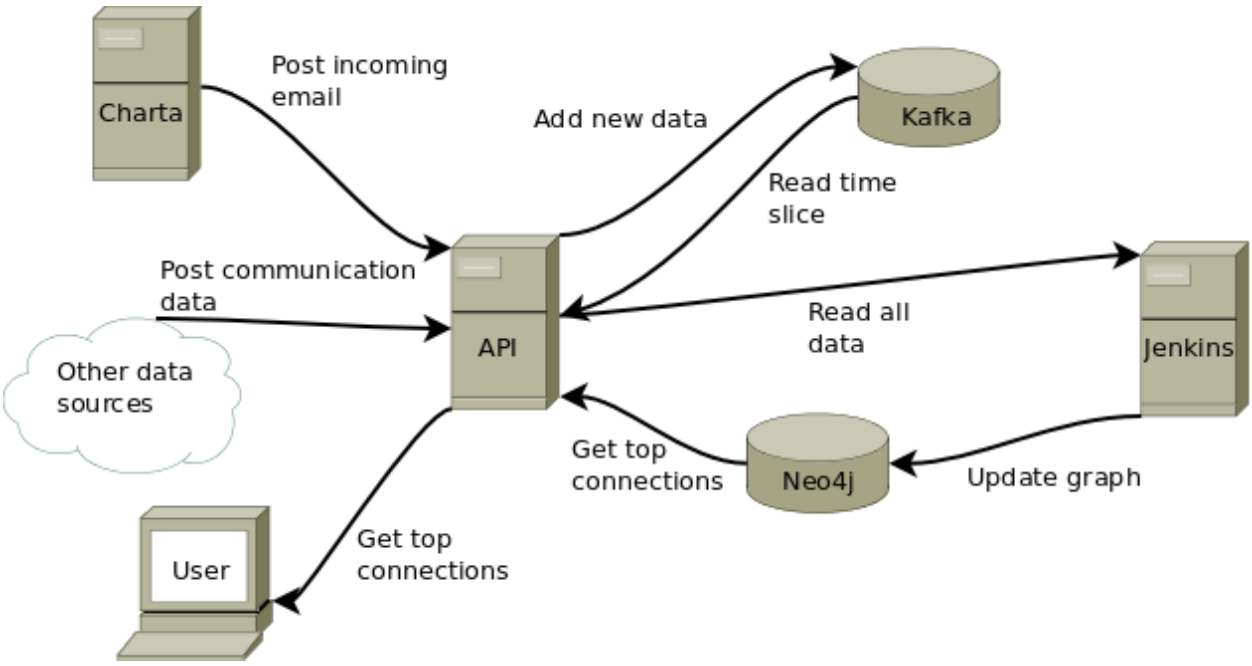
- Choose scalable technologies to allow for expansion to larger data sets.
- Write code in Java 8.
- Build project using Gradle.
- Use Jenkins for continuous integration.
- Filter out email from bots using the email-filter library.
- Store communication data in a scalable data store.
- Store top connections on a scalable graph server.
- Use DropWizard to set up an HTTP API.

IV. Project risks

The most important risk involved with our project is more of the ethical issue of the sharing and use of others' information. It is not only important to protect the information of others but to respect their privacy as well in its collection. In this project we are using sample data from ourselves and willing volunteers within the company. However, when the project is applied on a much broader scope, it is important that data is collected and used ethically. In some cases, the bodies of e-mails were redacted, but in other cases they weren't. Having access to that much information can be dangerous, and so the team needed to first confirm with the users that it was acceptable for us to be able to see this information, and then use it in a responsible manner at all times.

There are many technical risks associated with our project as we are using many tools that we were completely unfamiliar with, like Gradle and Kafka. In addition, we are using a myriad of third-party software for the purposes of creating graphs and determining an accurate score for a communication relationship. Most, if not all, of these libraries are very new (have been developed in the past 5 years), and so there is a very real possibility of bugs in the code. Not enough tests may have been run using the code, and so when put under the stress of a large code base with massive amounts of data, there is a possibility that the system just won't hold up. Many times the team ran into problems where known bugs occurred and there simply was not a fix for them. While most of these problems were minimal within the scope of the project, adaptability became a common theme early on in development.

System Architecture



Data Flow

Our main data source is currently Charta, an API which interfaces with Gmail IMAP. Through Charta, we have registered web hooks for 13 different email accounts which are called whenever an email is received. We extract the relevant data and forward it to the /add endpoint.

Email data from Charta, along with any data from other sources (currently only manually submitted), enters the main process through an API endpoint, /add, which takes JSON formatted data about a communication, including an actor, action, targets, and any additional attributes. This data is then timestamped and posted to the Kafka broker, which stores the data in a log. These logs persist for a configurable period of time before expiring. For our purposes, we decided that 90 days would be a reasonable period.

A Jenkins job periodically reads the communication data from Kafka, assembles it into graph form, and condenses it into a graph of connections between people. This data is then uploaded to the Neo4j graph database. To prevent downtime during this process, all data is labeled with a version number. Queries which read data first read the current version number, and only request data with this version number. The version number is updated when a new data set has completely finished uploading, at which point the old one is deleted.

The data in Neo4j can be accessed through the API endpoint /top. Data is returned as a JSON array of ordered top connections for a given identifier. Because all computations regarding

connection strength are only performed when the graph is being updated, this is a simple query which runs in under 500ms end-to-end. For a detailed description and list of resources and endpoints of our API, see Appendix A.

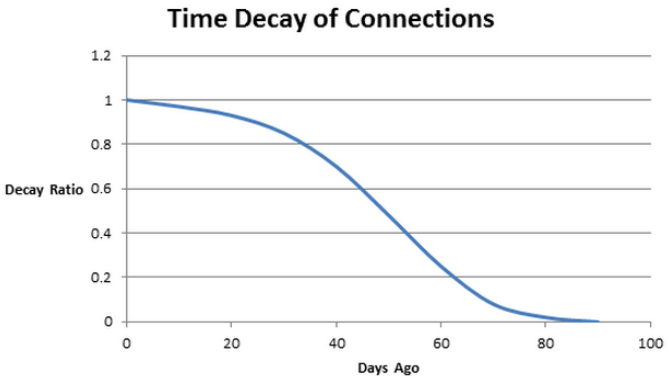
Technical Design

Graph Assembler

Determining the connectivity between two contacts takes place in two stages, assembly and analysis. The assembly portion consists of assembling a graph representing all the relevant communications made over a given time frame. In this graph, edges are not necessarily unique as all edges are kept as to preserve their data for the analysis portion.

The analysis portion in short takes the graph created in the assembly portion and condenses it down such that all edges between two nodes are unique. In this second graph, edges will represent two peoples' relationship rather than an individual instance of communication. Naturally, this edge will have a higher weight the stronger the connection is predicted to be, and this weight is determined using data preserved by the assembly portion. Using a variety of metrics, a raw score is determined for each relationship before it is normalized between zero and one hundred. The primary factor behind connection between two individuals is simply the message count. In short, if two individuals message each other frequently without a large discrepancy in the number of messages from each person, then they're likely to be connected. However, if this discrepancy reaches a certain relative threshold, it begins to decay the strength of the relationship. Given the scope of the project and our data, message counts were the driving force that caused people to be marked as connected, and all of the other data was generally used to look for special cases or to determine what messages should be viewed as more important.

Another way the team dealt with special cases for emails was the bot detection library, used to filter out bots using the email address. Obviously it is not acceptable for people to be marked as having a strong relationship with a spam bot so any address that was marked as a bot was disallowed from having a strong connection with any other contact. A secondary measure that was taken to prevent bot connections was checking message count discrepancy. Frequently bot messages are mailing lists and as such will send many messages and receive nearly none. To detect bot behavior we additionally cut connections where the messages in one direction were vastly greater than the other after a certain threshold of messages had been sent in total. A helpful side effect of this addition is it also filtered out instances of spamming or pestering between two human contacts.



The inclusion of the timestamp made our connections data alive as it allowed us to introduce time decay into our data. The older a message is, the less weight it ultimately has. Thus connections that contacted each other frequently in the past, but not so much recently, would receive a much lower connection score than two who contacted each other recently. This was intended behavior, as our generated connection scores were to represent the current amount of

connection. To further simulate the decay of connectivity, we used a logistic growth function. Using this function, all communications that were received somewhat recently received high weighting. However after a certain time period had passed, decay began to occur much more quickly until it begins to approach zero asymptotically.

In addition to all the previously mentioned used data that is essentially universal to all connection types, we made a distinction between "To:" and "CC:" specifically for email communications to help us further improve the accuracy of the algorithm. To put it simply, if a communication between two contacts was merely a CC, then the weight will be weakened by a constant multiplier to reinforce the strength of direct, personal contact.

System Scalability

Because our system could handle potentially massive amounts of data in the future, we had to choose scalable technologies. Although we are currently only handling a bit over one thousand incoming data points per day, this could easily rise to hundreds or thousands per minute if we incorporated more forms of communication and started collecting from FullContact Address Book users rather than just a small subset of FullContact employees who have supplied OAuth credentials to Charta.

Our front-end API operates as an auto-scaling group which automatically creates and destroys server instances depending on load. Right now, this cluster has its minimum and maximum instances both set to one, but just changing this configuration would allow the API server itself to scale and handle higher volumes of requests.

Expanding Kafka would be slightly more difficult and would require some additional development because of assumptions we made, but Kafka itself is designed to be scaled efficiently. Kafka brokers all connect to a central Apache Zookeeper server, which manages partitions and replication. In a larger system, each broker would likely contain a partition representing some portion of all the data. When retrieving data, each broker would send its portion of the data. We would need to add a way for the API server to keep track of the number of partitions and read from and write to each, but after that, this part of the system could scale indefinitely.

Condensing communication data into a graph of connections is not as time-sensitive as the actions behind the various API calls, but we still need to consider performance. We did some basic performance testing which showed processing times on the order of a few minutes for hundreds of thousands of communications. The real limiting factor in these tests was memory, which could be fixed in the short term by just adding more memory. In the long run, it might be worth reframing this as a distributed computing problem, possibly using something like MapReduce to condense potentially billions of communications down to a single connections graph.

Neo4j handles smaller data sets than Kafka because it stores connections between people, not every individual communication. It also only gets written to periodically when the graph is

actually updated. Because of this, read capacity is the only meaningful aspect for us to consider when scaling Neo4j, and it handles this quite well. Simply adding more servers to a Neo4j cluster linearly increases the read capacity.

Design Decisions

Language and Build System

We chose to develop the majority of our project in Java and continuously build it using Jenkins. Using Java allowed us to avoid the learning curve of a new language, while also giving us access to a powerful standard library and numerous external libraries. Gradle manages our dependencies and allows for extensive customization of different build tasks. Although there was a steep learning curve in the very beginning, Gradle now saves us time by automating the running of tests and creation of different artifacts. It also integrates with Jenkins, which automatically publishes new builds to our production servers. All of these technologies were already in place at FullContact, so choosing these programs was the right decision.

Data Store

We had to choose a scalable data store which allowed for easy insertion of individual pieces of communication data and retrieval of time slices of data. We initially considered Cassandra, MongoDB, Kafka, and HDFS, but eventually settled on Kafka. Kafka works well for our application because it scales easily by simply adding more brokers to the cluster, and the sequential log file structure allows easy access to slices of data. The disadvantage is that it is not meant for permanent storage, and messages expire after a certain time period, but since our analysis weighs older communications less, this should not affect our analysis much and actually solves the problem of continuously expanding the system to store more data.

API and DropWizard

Rather than interacting with our Kafka data store directly, we chose to implement a simple HTTP API to store and retrieve data. Although this added to our development time, it provides a convenient abstraction which makes it easier to gather data from other sources. For example, the business card team within the company can provide us with data by simply sending a POST request to the API with a JSON string representing the action of someone scanning a business card. Without the API, they would have to integrate a Kafka client into their own application. The API also gives us the flexibility of changing the specifics of the data store without changing how we interact with it. To specifically make HTTP requests through the API, we used DropWizard, a program already heavily in use throughout the applications of FullContact.

Charta and Email Classifier

For handling real-time in-bound emails, we decided on using Charta because the library was developed at FullContact. Therefore, we knew exactly how it worked, how it didn't work, and how to implement it (especially when we ran into problems). For handling bot detection, we also used another library developed at FullContact, the Email Classifier. Since both of these two were already in production by the same company for which we worked, it made sense to use the technology that was already in place.

Graph Modeler and Server

Tinkerpop is the most popular form of creating graph objects in Java today, and because of this, it became our clear choice very early on in development. There are a variety of methods already implemented by graph objects that allow us to get certain properties of the graph, like properties of edges and nodes, and our analysis algorithm directly uses most of these functions.

To meet the product requirements of having a remote server keep a graph of communication objects up to date, we utilized the third-party software of Neo4j. There are many ways to get a graph up to a server, such as OrientDB or the Rexster library within Tinkerpop, but we decided to go with Neo4j. It is typically seen as the most commonly used database system among graph libraries, it has plenty of support, it's efficient, and it was easy to implement in the context of our problem.

Graph Assembler

Though not entirely lengthy or complicated, the graph assembler took several iterations to reach its final state. Most of these iterations were to improve analysis by adding a new type of data, like cc's or timestamps. The team chose to allow assembly and analysis to be two different steps for a few main reasons.

First and foremost, it was simply better design to keep the two separate. Although maintaining only one graph conserves memory, coupling both tasks limits usability and makes analysis itself much more difficult. In addition, separating the functions allows different forms of analysis to be performed, an obvious must for the system to be scalable. How exactly to structure the two steps was one of the most pressing design issues as the project moved forward. At the onset, the tasks simply lived in their own function. Though this is the optimal design for the scope of our project, future iterations will likely include the addition of other forms of communication beyond emails. In this situation, the best design would likely include the use of an abstract class and subclasses each having their own analysis functions. Each communication in the graph would know how to calculate its own weight and all different types of communication could easily live together in the same graph. However, actual implementing this around Tinkerpop would be difficult and in this case it would mean abstracting before it is necessary. To create a healthy compromise between these two alternatives, assembly was its own function as before, but analysis was split. Every time an edge was analyzed, it called a different function based on the type of communication it was through the use of a switch case. In our situation, the only real case was email, but in this structure other types of communications could easily be implemented. This process separated and modularized the code rather effectively, so refactoring to use subclasses would be a much less painful process.

Another frequent point of design changes to the graph assembler was the actual analysis portion of the graph assembly. In truth, the graph analysis never had a final goal as it was impossible to find a definition of "done". Still, each iteration we improved the algorithm as its accuracy was critical to the effectiveness of the system as a whole. At the onset of the project, edges were weighted simply by raw message count, and from that point on, we constantly experimented with new ideas and new data. Overall, much more still remains to be explored with analysis that could make it even more effective.

Results

The goal of our project was to create an API which takes in communication data from multiple sources and analyzes it to determine connectedness between two people. Due to the scope of the project, this broad objective was narrowed down to only be used on emails. However, our design allows for use of other types of communication to be implemented relatively easily. Currently, our project has a remote Apache Kafka server running that accepts communication data conforming to a specified format and stores it. We periodically query this remote database and convert the received data into a graph which is then analyzed and condensed into a graph of connections between people. This graph is then pushed to a Neo4j server which can then be queried with a specific username to receive their "top contacts". Some of our greatest challenges stemmed from optimization as in a production scenario we will be working with massive data sets. The graph assembler's current performance times are displayed below.

Number of Messages	1000	10000	100000	250000
Assembly	0.026 s	0.133 s	1.217 s	4.481 s
Analysis	0.040 s	0.451 s	6.795 s	12.921 s

Originally, both assembly and analysis took much more time than they currently do. We decided it was imperative to make assembly and analysis as efficient as possible because the system absolutely had to be scalable. We believe there is still some improvement to be done in reducing the complexity of the analysis portion, although other tasks took a greater priority due to time constraints.

Another challenge with the graph assembler was the validation of the algorithm that determined the connection strength between two people. At first this was hardly testable as we lacked real data to use, and as a result the algorithm was fairly simple. However, due to the donation of a FullContact employee's inbox, we were able to test the algorithm against real data and receive feedback.

From the first testing cycle it was clear that too much weight was given in scenarios where people sent nearly the exact same number of messages to each other, and in fact important contacts were being filtered out as spam due to a large difference in raw message quantity. The algorithm was then rebuilt based on these observations, and the 2nd test cycle received very positive feedback.

Although the graph analysis we were performing seemed to be accurate, we decided to use timestamps of the email such that newer emails in the timeslice carried more weight compared to older emails. This was originally tested with linear decay, but it made little to no effect in most cases. In truth we believed that relationship strength shouldn't decay in a linear fashion. After some testing with various models, the team decided that a logistic function would be most

suitable to model the decay we were looking for. There was little to no decay for a certain time, followed by much more rapid decay at a certain age. Testing of this algorithm showed one particular contact decaying heavily due to age weighting. When asked about it, the owner of the inbox stated that they'd heavily contacted that person in particular in the past, but not recently. Due to this feedback, we continued to use the logistic model as it seemed to be accurate.

However, as more testing was done, we noticed bot accounts were sometimes receiving high weights. The addition of the Email Classifier essentially removed this issue, with an unfortunate side effect of occasionally filtering out people that weren't bots. Overall, when launched on real data the results seem to be mostly positive; one major issue is that Charta only reads incoming emails. So connections where only 1 contact has a web hook consist of communications from only one side, leading to heavy decay and often incorrect data.

Appendix A: API Documentation

The API is currently accessible at http://fieldsessionapi.fullcontact.com, and the GitHub repository with the complete README is available at https://github.com/fullcontact/field-session-2014.

POST /add

Adds the given communication data to Kafka, adding the current server timestamp. If the data is invalid, nothing is added.

Data

The data for a message object expects a JSON string containing the following fields:

- **actor**: String
  - Who performed the action? Use a unique identifier of some sort. Ex: For an email, use the email address.
- **action**: String
  - What action was performed? Ex: email
- **targets**: Non-empty Array
  - A comma-separated list of targets affected by the action. Ex: recipients of an email.
- **attributes** (*optional*): Object
  - Add additional metadata about the communication which might be useful in processing here. Not required. Ex: length of an email.

Example Usage

```
curl -v -X POST -d "{\"actor\":\"foo@bar.com\", \"action\":\"email\", \"targets\":[\"abc\"], \"attributes\":{\"length\":200}}"
http://fieldsessionapi.fullcontact.com/add
```

Response

- **202 Accepted**: The data is correctly formatted and was added to the data store.
- **422 Unprocessable Entity**: There was an error with the data format. The response body will contain more information.

POST /charta

Allows for the posting of data to the Kafka queue by adding a web hook to an e-mail event. While this endpoint is not aimed to be utilized by users, it allows for the addition of data to the server using real-time e-mail. To add users to the list of "tracked" e-mails, the RESTful API server interface must be updated with the id of the e-mail address.

GET /get

Returns a JSON array containing communications in the given time range.

Parameters

- start -

Returned values will have timestamps greater than or equal to this value, expressed in epoch milliseconds. Defaults to 0.

Example value: 1401470011229

- end -

Returned values will have timestamps less than or equal to this value, expressed in epoch milliseconds. Defaults to current time.

Example value: 1401470081697

Example Usage

curl "http://fieldsessionapi.fullcontact.com/get?start=1401470043524&end=40147008169"

Response

The response body contains a JSON array of communication data. Example:

[{"actor":"foo@bar.com","action":"email:to","timestamp":1401470043524,"targets":["abc"],"attributes":{"length":"200"}},
{"actor":"foo@bar.com","action":"email:cc","timestamp":1401470053366,"targets":["abc"],"attributes":{"length":"200"}},
{"actor":"foo@bar.com","action":"email:to","timestamp":1401470063508,"targets":["def"],"attributes":{"length":"200"}},
{"actor":"foo@bar.com","action":"email","timestamp":1401470073545,"targets":["ghi"],"attributes":{"length":"200"}},
{"actor":"foo@bar.com","action":"email","timestamp":1401470081697,"targets":["jkl"],"attributes":{"length":"200"}}]

If no data exists on the server for the given range, an empty array is returned.

GET /top

Returns a JSON array of a specified number of top connections for a given identifier.

Parameters

- id -

The identifier of the person being queried. Currently, contacts are not condensed, so providing an email address will only give information about email contact. This should change in the future.

- count -

The number of top connections to return. Defaults to 5.

Example Usage

Curl -v "http://fieldsessionapi.fullcontact.com/top?id=scott@fullcontact.com&count=5"

Response

The response body contains a JSON array of connections along with their connection strengths. Example:

[{"id":"katy.hammond@loqate.com","connection":7.333333333333334},
{"id":"kaspars@fullcontact.com","connection":7.178130511463845},
{"id":"jeanette@fullcontact.com","connection":6.8789808917197455},
{"id":"jaclyn@fullcontact.com","connection":6.8040147913365026},
{"id":"john@fullcontact.com","connection":6.616541353383459}]

POST /charta

Accepts application/json data with a structure defined in ContextIoWebhookResponse. This endpoint is designed to receive incoming messages from Charta. It strips out the required information and creates a Communication object, which is then added to Kafka.