The Book of Myriad

Volume I

Documentation for courageous Developers

Contents

Introduction

Hey, thanks for taking the time to look into Myriad's Developer Documentation. We chose to write a documentation for this software because, against our earnest efforts, it is not a trivial piece of software. We're not saying that to warn you, or to scare you – but, knowing ourselves, we can truly understand the urge to just look at the code and go "bah". Writing an email client is an inherently ugly process as it deals with legacy technology and uses ambiguous concepts like "message" or "response". So thanks for your patience.
In return, we promise that only a couple parts of the system are ~~utterly stupid~~ complicated.

— April 2013, Ludwig, ludwig@cs.stanford.edu

TL;DR: We apologise for the inconvenience.

PS: In truly bad cases, I'm positive we'll be available for a Skype interview on the code. Just promise to update this document with the things you learned, and we'll be happy to help you.

Technologies

Technology Stack

Myriad is a web application that is accessed by users via their browser. Myriad runs on one *ubuntu* server that provides a web server (*nginx*), a database (*MySQL*), a key-value store (*Redis*) for enqueuing background jobs (defunkt/*resque*), a *Rails* App Server (Phusion *Passenger*), timed maintenance tasks (*cronjob*) and process monitoring (*monit*). The source code is stored on *github* and can be deployed with an included deploy script for *capistrano*.

Our Rails Stack uses *OmniAuth* for authentication, *jquery* for javascript, *thin* as a developer webserver, *bootstrap* as a CSS framework, *SASS* for better CSS, *resque* for background jobs, *mail* for Email parsing (with custom additions), *better_errors* for nice developer error messages, *backup* for, well, db backups, *pry* as a better debugger, and unfortunately no tests.

The tech stack was created by trying to use the most commonly used components as of the end of 2012. We choose simple, everyday components with plenty of documentation. Most solutions were taken straight from railscasts.com, ruby.railstutorial.org and destroyallsoftware.com so you should be able to look those up if you need to reverse engineer our thought process.

For your development machines we recommend ubuntu or Mac OS X. On a Mac, you'll want to become familiar with the Terminal and a package manager like `homebrew`; ubuntu users will already know `apt-get`.

Remaining inconsistencies may be due to the fact that we started documenting late, and learned a lot during this project ourselves. So if you see something that is way more complicated than it should be; don't be afraid to change it! We might not have known any better at the time. Thanks for understanding.

Our Git Strategy

…is loosely based on a simplified version of *gitflow*.
We deploy from *develop*, which contains our most current semi-stable source. Features are developed on *feature branches* whose names start with "feature/", e.g. "feature/email-attachments". When a feature is complete, we open a Pull Request into *develop* on github. After a code review a different developer accepts the Pull Request. Then, all

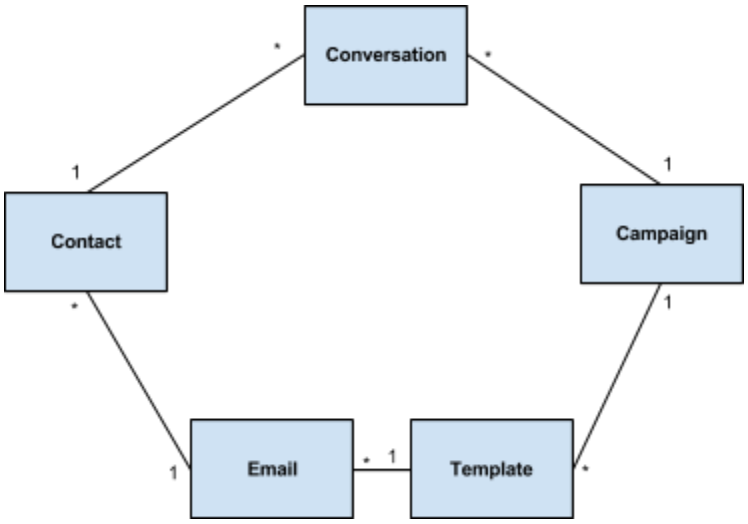
developers merge the updated *develop* into their current *feature branches*, and the merged *feature branch* is deleted. We found this to be the minimum requirements for a clean repository with just enough structure to catch the majority of bugs early enough. Your team might have different requirements, but we don't recommend using significantly less structure.
If you set up a production server, you might want to start using *tags* on *master* for releases.

Classes

Core Classes

Myriad is built around 5 model level concepts, or classes:

1. Contacts
2. Conversations
3. Campaigns
4. Templates
5. Emails



Contacts

Contacts are the names and email addresses of the people you want to email.
They are created when importing from spreadsheets, when typing an email address into a To: field, or when importing existing emails by adding a Gmail label to them. Contacts belong to users. This means that there is only one fred@gmail.com for each user and this fred@gmail.com may be linked to multiple campaigns of that user.

Templates

Templates are prototypes of emails. They consist of body text that can contain `placeholders`, a subject, actions that are triggered when they are sent, and searches/rules that can send them automatically.

Emails

Emails are instantiations of Templates. Their body doesn't contain `placeholders` anymore, but only the merged email body. They have a delivery status, for example, and keep lots of information on their external representations, like a message_id, UIDs for IMAP folders, thread IDs (THRIDs) for Gmail, etc.

Campaigns

Campaigns are a collection of `templates` and meta information like a connected Google Drive `spreadsheet`. They additionally contain a list of `Keys`. You can think of keys as column headers in a spreadsheet.

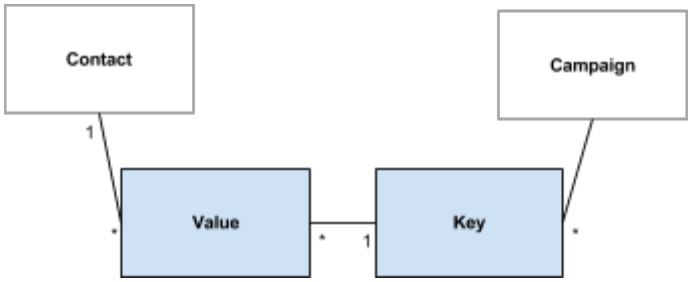
Conversations

Conversations are like email threads in gmail, but scoped to a specific `campaign` and `contact`. They contain all the `emails` a `contact` wrote or received within a `campaign`.

Further Classes

Keys and Values

One goal of a campaign can be the collection of information. For that, users create a data schema made up of `Keys`, that they (or somebody they share the campaign with) fill with `Values`. The Keys consist of a *name* – the header of the spreadsheet column – while values have a *content* – the content of a spreadsheet cell in the row of the contact they belong to.

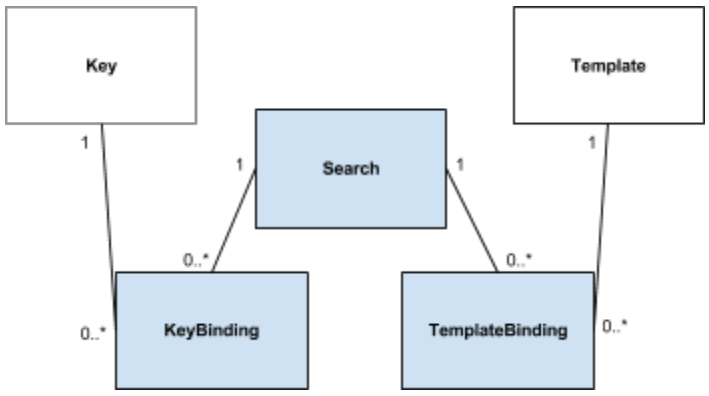


Search & KeyBindings

Searches allow users to constrain the set of all conversations to a subset of them by defining criteria those conversations match. Each search consists of zero or more KeyBindings, zero or more TemplateBindings, as well as an optional conversation status constraint. A KeyBinding is a value that a key must have to satisfy a specific search. Similarly, a TemplateBinding *tb* of search *s* specifies a Template *t* such that a conversation *c* matches *s* if *c* contains an email *e*, and *e* is an instantiation of *t*.

For example, a search might specify that a conversation should be unread to be part of the search's results. But we also want to allow users to specify constraints on their own Keys, e.g. coming? = "yes". For that, we needed a flexible way to specify those constraints. So we came up with the concept of a KeyBinding; it "binds the free Key variable to a specific value". So a KeyBinding associates a search with a Key that contains a specific value (todo: add comparison operators. e.g. age > 21).

Note that Searches are saved to the database. If they specify a template to be sent, we consider them to be "Rules" that can be automatically triggered. Otherwise they just clutter up the database and should eventually be purged by a cronjob. This persisting of searches could be used to remember recent searches.

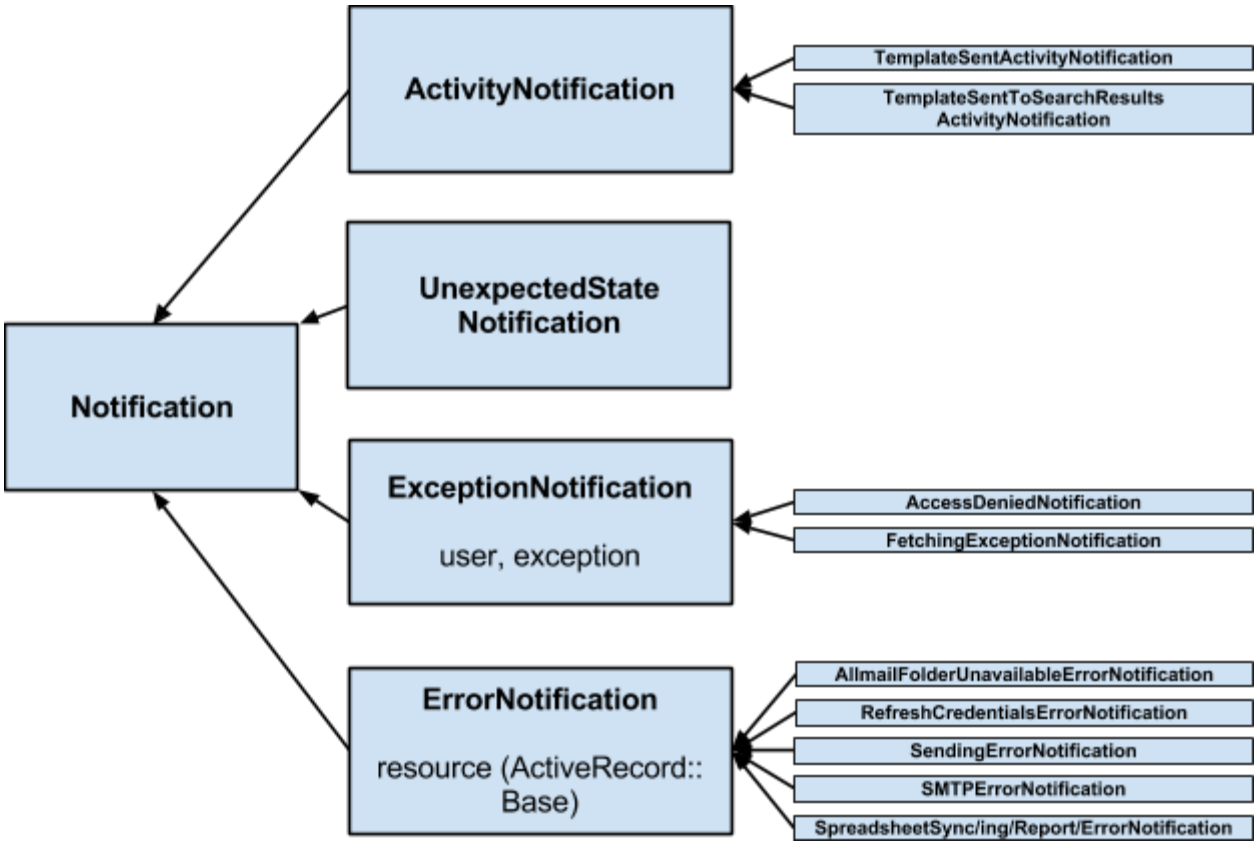


Notifications

Notifications are used to "notify" but not only. They also hold state (e.g. invalid user credentials) that can be used only internally by the system (e.g. to avoid syncing emails from users with invalid credentials). They are like "state" descriptors of other objects and in some cases these are visible to the users (e.g. we are syncing your spreadsheet), in other cases only the backend uses them. So maybe they should be called differently. (any ideas?)

They're an abstract class that we might have taken a little too far. They basically contain a polymorphic *Type* field, and an *ID*. Then, madness ensues as we use this information to attach Notifications to *any ActiveRecord::Base object in our database*. There is a whole class hierarchy beneath `Notification`, containing classes like `ErrorNotification`, which uses aforementioned feature to attach Errors or warnings to basically anything. There are also very specific subclasses like `TemplateSentToSearchResultsActivityNotification`, which are used to notify users. Another example would be a `SpreadsheetSyncingNotification`, which is also used to inform users. You can define groups of Notification Subclasses that are displayed in different parts of the UI, for example on the Admin Dashboard or on top of the screen for the respective user, as a kind of persistent flash message.

The Notification Hierarchy



ActivityNotification

These are used to display the activity stream, which is currently woefully underdeveloped.

They could also be used to record Assistant actions in the future. Maybe they could even be tied to resources to enable tracking of the last spreadsheet sync, etc.

UnexpectedStateNotification

We used these to make sure that all of those little instances where you're writing a case statement without a default, expecting values to be of a certain type, etc *actually* never occur. And in reality, of course, they do. So if you're almost certain a certain state should not be reached in a program, put an `UnexpectedStateNotification` there and be greatful when, two months later, your admin dashboard informs you that what you had deemed impossible happened 6,000 times last night.

ExceptionNotification

A simple way to "save" an exception for review from the Admin Dashboard. (Todo: Also save the stacktrace with these, so they might be even more useful.)

ErrorNotification

These might be named a little badly, since they can actually mean *anything that is attached to a certain resource and might be resolved later on*. For example, we could use these to implement an `InvalidCredentialsErrorNotification`. When, at a later point, the credentials work again, we'll be able to call `InvalidCredentialsErrorNotification.resolve` and pass it the user object, deleting the `InvalidCredentialsErrorNotification`.
An example where these are used to notify not about an error, but simply about a state that's attached to a resource is the `SpreadsheetSyncingNotification`.
A possibly better future name might be *PersistentNotification* or *ResourceNotification*.

Notification Groups

`notification.rb` defines two constants of Notification Subclasses, `ACTIVITY_NOTIFICATION_TYPES` and `IMPORTANT_NOTIFICATION_TYPES`, as well as a scope `:admin`. These are used to decide which Notifications are displayed where.

ACTIVITY_NOTIFICATION_TYPES

…are displayed only in the Activity Stream. If you want custom Notification Types to be displayed there, add them to this Array.

IMPORTANT_NOTIFICATION_TYPES

…are displayed at the top of the screen, like a persistent *flash*.

Scope :admin

…includes every Notification subtype but the `ACTIVITY_NOTIFICATION_TYPES`.
This is intentional, so that any notification that is displayed to users is also brought to the attention of admins. As the system becomes more predictable you might want to remove the `IMPORTANT_NOTIFICATION_TYPES` from this as well, and rename them to `USER_NOTIFICATION_TYPES`.

Automatic Display

If you take a look at layouts/notifications.html.haml, you will notice we use a little trick to display your notification automatically if you provide a template under `app/views/notifications/` with the class name of your notification underscore'd. So a `SpreadsheetSyncingNotification` has a view called `_spreadsheet_syncing_notification.html.erb`. This works for both `ACTIVITY_NOTIFICATION_TYPES` and `IMPORTANT_NOTIFICATION_TYPES`.

ValueSettingActions

This one sounds worse than it is. Similar to a KeyBinding, it connects a Key with something else while providing a value. Unlike a KeyBinding, however, a ValueSettingAction isn't used in *searching* for this value, but in *setting* this value. A Template can have multiple of those, which are used after this template is sent to set the specified Key to the contained Value of the ValueSettingAction.

The Email subclass hierarchy

Email is hard. Myriad differentiates emails along two dimensions: whether they are fetched from an IMAP server or created by the Myriad server, and whether they are semantically outgoing emails – i.e., an email a user has sent to a contact – or incoming emails – i.e., a reply from a contact to the user. Since not all user emails necessarily start in Myriad, this differentiation is vital to discern between incoming emails, and emails that users simply sent from their Gmail Accounts.

Emails	**Fetched**	**Created**

Incoming	IncomingFetchedEmail Replies from contacts	~~IncomingCreatedEmail~~ *apart from mocking, none*
Outgoing	OutgoingFetchedEmail Emails sent by the user on Gmail	OutgoingCreatedEmail Emails sent by the user on Myriad

A nice little special case is an `IncomingFetchedEmail` subclass called DeliveryStatusNotificationEmail, or `DSNEmail` for short. It is used to discern Mailserver generated delivery status notifications (which usually indicate a delivery failed error) from normal incoming emails. `IncomingCreatedEmail` is not currently used, but might prove to be a handy concept when mocking incoming emails for user studies.

Case studies of class interaction/design decisions

KeyObserver, Key, Exception-/ErrorNotifications

Keys act as a representation of Spreadsheet Column headers, so when a user renames a Key, we also want to rename the respective spreadsheet column. Since an inconsistency could lead to data duplication and general confusion, we *do not* use a background worker to set the value. Since the functionality of the Key class is not directly dependant on the Spreadsheet class, we decided to put this functionality into an external observer, *KeyObserver*. Here's the code of the after_update callback:

```
1:  def after_update key
2:    return unless key.campaign.spreadsheet.present? and key.name_changed?
3:    begin
4:      key.campaign.spreadsheet.rename_key!(key.name_was, key.name)
5:    rescue Exception => exception
6:      ErrorNotification.add key, exception.message
7:    end
8:  end # after_update
```

Let's walk through this code. Line 1 and 8 define that this callback method, which will be called after a *Key* object is updated. The second line acts as a guard, returning without doing anything, unless the Key object belongs to a campaign that has a spreadsheet and it's name was changed. The name_changed function is one of those dynamically generated magic

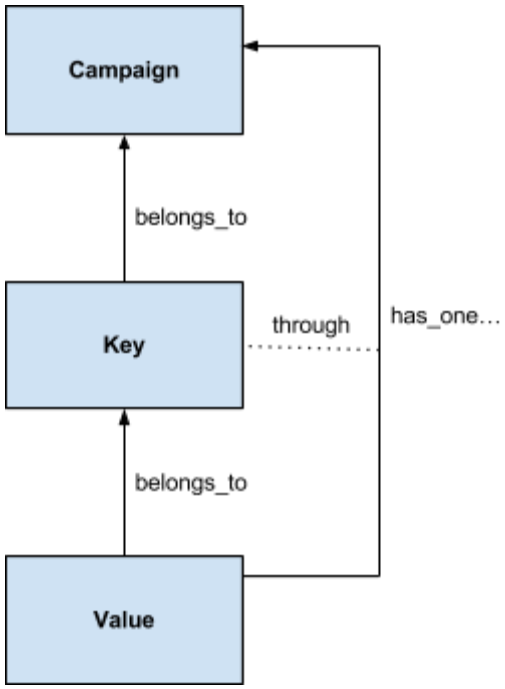
ActiveRecord methods. It's easy to understand what it does, but hard to guess it even exists. When you encounter such methdos, try googling for them without the object specific part, e.g. *_changed?*. Same for the *_was* method in line 4.
Line 6 uses our custom ErrorNotification class to give admins a record of the exception. Here we used an ErrorNotification instead of an ExceptionNotification because we want to attach it to the Key object. (*ErrorNotification.add key …*) ExceptionNotifications are used in places were we don't know which object caused the exception. Also you can't attach an ErrorNotification to a non ActiveRecord::Base Object, as we use the class name and ID to identify the object.

Authentication with CanCan

I'm sorry, Dave. I'm afraid I can't do that.

— HAL, 2001: A Space Odyssey

CanCan uses a file called ability.rb to define what certain users are allowed to do. It uses a nice, but very limited DSL for this. We introduce a simple one-liner block and a #authenticate(user, resource) method to simplify authentication in Myriad.
The authenticate method relies on the fact that most resources are directly or indirectly associated with a campaign.



For example, a `Value belongs_to Key` that `belongs_to Campaign`, so a `Value has_one campaign, through :key`. That's true for most resources.

There are resources like User, which don't belong to a campaign. Those are authorized in the

traditional CanCan way. See ability.rb, lines 16–18.
For all other (nested) resources; our approach simplifies the authentication: a line like `can :manage, Key, &authenticate_user_for_resource` passes the resource to be authorized into the block `&authenticate_user_for_resource`, which was earlier defined as `{ |resource| authenticate user, resource }`, eventually passing the resource to the authenticate method itself.
To authorize a new Resource that belongs to a campaign; say a `Puppy`, you'd thus simply write:
`can :manage, Puppy, &authenticate_user_for_resource`
As long as your Puppy class `has_one campaign, through :whatever`, this will work.
You could imagine taking this even further by defining a custom superclass for our resource classes, that would automate authentication entirely… a boy can dream, can't he?

Email Fetching

Learn yourself an IMAP for great good.
— Christian "1 EXAMINE 17 2 EXISTS 2 OK" Ikas, ikas@stanford.edu

Fetching emails from Gmail - a brief overview

As we spent a lot of time figuring out the intricacies of the Gmail IMAP implementation and the manifold cases that can occur when normal (or heavy) users go about their daily email business without considering the restrictions of poor Myriad, we want to spare you repeating the same.
We hope to have brought the fetching and email creation logic to a relatively stable state, yet we are certain there remain a bunch of holes that could cause failures in edge cases.
If you do not need to touch the code, you might find the following explanation too detailed, so you shouldn't waste your time with it (bar this overview section). For anybody working with the fetching code, however, the following information should prove very useful - especially if you are unfamiliar with email headers, IMAP protocol details and the like.
In brief, our fetching process works as follows:

1. **Identifying emails on the Gmail server:** First, we connect to the user's Gmail account and identify the UIDs (one of many types of email identifiers, see below for a more detailed explanation) of all emails we want to fetch. We use multiple different queries for that, which will later be explained in more detail. Before we move on to the next step, we filter out emails in our database whose UIDs we already know, i.e. have fetched before.
2. **Fetching and processing email data:** For the remaining UIDs, we fetch the

corresponding email from Gmail, create a Mail object (*from the mail gem*) for it and determine its type – IncomingFetchedEmail, OutgoingFetchedEmail, etc.

3. **Email creation in Myriad:** As a last (but not trivial) step we try to use the information from the Mail object to create an Email object in Myriad.

That was very general, so let's move on to the gruesome details.

Background information on Email identifiers

This is the way society functions. Aren't you a part of society?

— Kramer, *Seinfeld*

In order to make sense of the following explanations, it is important to know about the myriad of different identifiers used for identifying emails (or threads of emails for that matter) in different situations. Some of these identifiers are part of the email or IMAP standards, some are Gmail-specific, some are globally unique, some are not, and so on.

Message-ID - looks like this: *95c7557e-ac98-11e2-949d-001e68f44ee6.65.1@myriad.cs.stanford.edu*

Each email has a Message-ID as a globally unique identifier (as specified in its header). This is part of the email standard (see RFC822, RFC2822, RFC5322 for further information). In order to make a clear distinction from other types of identifiers, in our code and database we usually refer to message-IDs as *header message ids*. When an email is sent out via Myriad, the generated header message ID contains a Myriad- and user-specific identifier. Gmail will respect this header message ID and pass it on when sending out the email (provided it is valid).

Note: Since Message-IDs are created by user email clients, there is no guarantee that they are actually globally unique. A faulty email client could potentially mark all outgoing emails with the same Message-ID. (Possibly that's why we need so many identifiers for emails in IMAP.)

Closely related is the "in-reply-to" field in the email header, which specifies the message-ID of the email that the current email is in reply to. This field can be missing if the current email is not in reply to anything.

Unique identifier (UID) - looks like this: *120* or this: *63829* (depending on mailbox size)

Part of the IMAP standard (see RFC3501), a UID should be unique and persistent for an email within a certain mailbox (aka folder). If an email is expunged/deleted form a folder, its UID *should* never be reassigned to any other email in that folder. Theoretically, there is a mechanism for detecting if such reassignment has taken place (again see RFC3501), but we never ran into this with Gmail and we are pretty sure they stick to this recommendation. Yet, if two mailboxes contain the same email, e.g. because the email has two labels (which are treated as folders) in

Gmail, this email will have differing UIDs in those folders (see "The importance of the Allmail Folder"). We make extensive use of emails' UIDs for fetching purposes and also store them in our email database table.

But wait, there's more.

Sequence number - looks like this: *50* or this: *2392* (depending on mailbox size)

Specifies the relative position of an email in a mailbox/folder. This position may change (e.g. when other emails in that folder are deleted), so it is not suitable for identifying emails across sessions. We do not use this identifier.

Gmail message ID [X-GM-MSGID] - looks like this: *1433265886521969447*

An additional ID introduced by Gmail that identifies emails across folders. We currently do not use this id (but might in the future, see "Future improvements").

Gmail thread ID [X-GM-THRID] - looks like this: *1433161678639370049*

When grouping emails into threads/conversations, Gmail assigns the respective thread identifier to these emails. We make use of emails' thread identifiers for fetching purposes and also store them in our email database table. We noticed that the the thread ID tends to coincide with the message ID of the very first message in a thread.

The fetching process — step by step

> *Oooh, ahhh, that's how it always starts. Then later there's running and screaming.*
>
> — Ian Malcolm, *The Lost World: Jurassic Park*

Step I: Identifying emails on the Gmail server

First result on Google Images for “Fetching Modes”

Fetching modes

As Myriad is only interested in a subset of the emails in a user's inbox, the process used to identify exactly which of those emails are relevant and what to do with them is crucial. Like the rest of the project, the fetching logic underwent a series of evolutionary steps. With each of those steps, we tried to make fetching smarter and more convenient for the user. Still, it is very much a work in progress. Currently, we use three different criteria to decide whether we want to import an email into Myriad: its *in-reply-to identifier*, its *Gmail thread ID* and any potential *Gmail labels* applied to it. We use all three modes because each of them is capable of picking up emails that the other two might miss.

Reply mode

We started out by simply querying Gmail for any emails that are in reply to an email carrying a Myriad identifier. This is the basic case and lets us collect all emails that are responses to anything a user sent out via Myriad.

Conversation mode

It happens frequently that users respond to Myriad conversations right out of their Gmail inbox. This caused problems for Myriad: It neither knew this email (as Myriad did not create it itself) nor would it pick up on any potential replies to this email (because they would be in-reply-to a message-ID identifier generated by Gmail, not by Myriad).
Fortunately, the thread (aka conversation) identifier assigned to each email by Gmail provided a means of tackling this problem. When fetching emails, we fetch this identifier along with other data and store it in our database. It makes us aware of this conversation on Gmail, and from now on enables us to search for other emails belonging to the same conversation. This means it is sufficient for Myriad to know a single email in a conversation in order to pick up on all other emails in that conversation.

Label mode

Up until a certain point, we lived by the principle that a conversation always has to originate in Myriad (in form of an OutgoingCreatedEmail) in order to be handled by Myriad. After a while, we received feedback from users who wanted to import already existing conversations into Myriad. This could be conversations initiated by the user from within their Gmail inbox (in which case Myriad neither knows the thread ID nor is able to detect its own in-reply-to identifier or it could be

incoming emails (e.g. recurring, repetitive requests the user wants to handle via a Myriad campaign).
In order to enable this import, Myriad fetches emails that have been assigned certain labels in the user's inbox. As to ensure the right labeling format, a corresponding campaign label is created automatically by Myriad whenever label syncing is activated for a campaign. All campaign labels are created using the campaign's slug and nested below the label "myriad" in the user's Gmail interface (mainly to avoid cluttering up the their label structure). This way, a campaign called "Test Campaign" will receive the label "myriad/test-campaign". Please see the section "Gmail quirks" for more detailed information about Gmail label handling.

Restricting our searches

Time limit

For large inboxes, querying Gmail for UIDs matching certain criteria can take a long time. In order to improve efficiency, we restrict our queries to a certain timeframe. The current default is to only search emails from the past two weeks. If necessary, this default can be overridden by passing any other time limit to the MessageFetcher.

Ignore drafts and certain labels

Email drafts are stored by Gmail as normal emails labeled as "draft". In order to prevent the fetching of drafts into Myriad, we ignore all emails with this label.
Especially when debugging the Fetcher, it may come in handy to prevent other emails/conversations from being fetched as well. To achieve this, assign the label "myriad_ignore" to those conversations in the Gmail interface. When viewing the conversation in Gmail, you can assign a label by clicking on the label icon above the actual conversation (right below the search bar). There you can either choose an existing label or create a new one.
Note: You can specify further labels to be ignored by Myriad in the Gmail config file (gmail.yml).

Step II: Fetching and processing email data

> "If we knew what we were doing, it would not be called research, would it?"
>
> *— Albert Einstein*

Fetching

After retrieving all relevant UIDs, we try to match them with the UIDs we have in our database and eliminate those we already know. We then fetch the actual email from Gmail, along with

some metadata (the UID and THRID as well as the set of labels assigned to the email). We use the email data to create a Mail object. All Mail objects created in the same run are sorted chronologically (oldest first) in order to ensure we can recreate the in-reply-to chain later.

Identifying email type

Before we pass on the mail data + meta data to email creation, we determine the applicable subtype of the FetchedEmail class.
Case 1 - **OutgoingFetchedEmail**: The Mail object's *from address* contains any of the current user's email addresses. We assume this is an email sent out by the user via Gmail.
Case 2 - **IncomingFetchedEmail (DSN)**: We try to detect Delivery Status Notifications as a special case of IncomingFetchedEmails. The Mail gem seems to not pick up on Gmail DSNs (possibly due to Gmail not complying with certain standards). We extended the Mail class in order to at least pick up on Delivery Failure Notifications. We seem to be able to display them to users pretty reliably, yet any other kind of Status Notification is untested.
Case 3 - **IncomingFetchedEmail:** If the email was matched in neither of the previous two cases, we assume it is a normal incoming email from a contact.
Now on to creating the email.

Step III: Email creation in Myriad

Basic attributes

First we create a new email object and set some attributes that we can copy from the Mail object very straightforwardly (e.g. *to email address*, *from email address*, *subject*). We also persist the UID and THRID (that we passed as metadata) within the Email object.

Detecting the email's predecessor (aka in_reply_to)

Myriad now tries to detect which email this new email is in reply to. We are interested in this information because it makes it easy for us to identify which campaign and contact (and as a result which conversation) this new email should be assigned to.
After making sure this email is not a Delivery Status Notification (this would require special treatment), we examine the *In-reply-to* header field, which contains the message-ID of the new email's predecessor. This field will only be present if the new email is in reply to another email. If we were able to retrieve a message-ID, we will look up the corresponding email in our database and - if successful - set the identified email as the new email's predecessor (or

in_reply_to_email, as we call it in the code).
If this process fails at any step (i.e. there is no *In-reply-to* field in the header or we are unable to find the corresponding email in our database), we move on to checking the *References*-field in the email's header. The *References*-field contains a list of message-IDs that (potentially) make up the complete reply-chain up to the current email. We scan this chain in search of the most recent email we know and - if successful - set the identified email as the new email's in_reply_to. If this also fails (because there is no *References*-field or it does not contain any known email), we do not assign the new email an in_reply_to_email.
Background: Checking the *References*-field enables us to capture emails that were not responded to directly, e.g. because they have been forwarded along the way. An application of this could be the following: A *Myriad user* sends an email to a *recipient*. Instead of replying directly, the *recipient* forwards this email to their *assistant*. The assistant then responds to the *Myriad user* on behalf of the *recipient*. The last email (as received by the *Myriad user*) is now in in-reply-to the second email (forward from the *recipient* to the *assistant).* As Myriad does not know of this email, it would now be the missing link that throws Myriad off. If the first email (sent by the *Myriad user*) is still listed in the References-field, however, Myriad is able to properly associate the new email with the one it sent out earlier.
Note: The way the References-field is handled is not consistent among email clients. Some may cap the list at a certain length and some may not set it altogether. Some preserve it when an email is forwarded, some don't. Hence, the References-field offers us a bonus chance to detect where a certain email belongs but it is nothing we can rely on in any way.

Campaign and template

We now associate the new email with a campaign. It is important to understand that not every email is affiliated with a template but every email has to belong to campaign. If we were able able to detect the email's predecessor in the previous step, we will now check whether this predecessor has a template. If yes, we can assign the new email the campaign associated with this template.
If we were not successful in detecting the new email's predecessor or if the predecessor does not have a template, we will now look at the labels that we got handed as part of the new email's metadata. We scan those labels for Myriad campaign labels (in the format *"myriad/campaign-identifier"*, where *campaign-identifier* is the campaign's slug). If we find a label that matches a Myriad campaign, we will set this as the email's campaign. If we were unable to

identify the campaign even in this step, we are at our wit's end and have to discard the email.
Note: This is where emails which we mistakenly picked up on should fail at the latest. Example: If you use your Gmail account both for the production server and your local server, one server will pick up on emails that are replies to emails bearing a Myriad identifier sent from the other server. This kind of false positive will go a long way in the process until we realize that we do not know what to do with it. If you ever get weird entries in the fetching log, consider this case. Deleting emails in your local test database or in your Gmail account might also cause (at first inexplicable) errors/warnings during fetching.

Contact

As a last step before we can finally save the new email, we will have to assign it a contact. If we were able to detect its *in_reply_to*, this is easy - we can just assign it the same contact as its predecessor. If this email is not in reply to any email we know, we check whether the current user already has a contact with this *from email address*. If they do, we set it as the new email's contact. If not, we will have to create a new contact based on the sender information of the new email (email address and potentially first and last name). Done!

Workers

We use workers to idle on a user's Gmail account, fetch emails, send email, fetch UIDs/THRIDs and set labels.
Note: Whenever you need to debug one of the workers, it might be helpful to look at the respective log in the log folder.

MessageFetcher (find the log at *myriad/log/message_fetcher.log*)

The MessageFetcher worker triggers the fetching process described above. There generally are four ways jobs are enqueued: 1) We fetch a user's emails whenever they log in. 2) We fetch whenever the user's idle thread received a notification from Gmail. We only idle for users that have logged in in the past week. 3) Once a day, we enqueue a full fetch for all users (also those which haven't logged in in a long time). 4) A user can manually trigger fetching in the *edit campaign* screen. It is also possible to trigger message fetching manually from the rails console (see "Useful rails console debugging and maintenance commands").

IdFetcher

When fetching emails, we fetch their UIDs and THRIDs along with the email data and store them

in our database. For emails sent out via Myriad, we do not know their UIDs/THRIDs. As we still want to retrieve them (e.g. to set a potential campaign label on Gmail for them), we trigger the IdFetcher to do just that right after sending an email from Myriad. When called, the IdFetcher sleeps for ten seconds in order to give Gmail sufficient time to create the email.
Note: If we were not able to successfully retrieve the ids within three tries, we quit and never retry. Currently we use rake tasks to ensure we still get them eventually (see "Rake maintenance tasks").

MessageLabelSetter

We fetch emails that are assigned one of the user's campaigns' labels. To provide consistency and increase user awareness of which conversations Myriad is handling, we also apply the respective campaign label to OutgoingCreatedEmails. In order to prevent cluttering up the user's Gmail web interface, we nest all campaign labels under a common label: "myriad".
Note: The setting of labels for OutgoingCreatedEmails only occurs if Label Syncing is activated for that campaign.

CampaignLabelCreator

To allow users to label their email for import into Myriad, we need to create the corresponding campaign label in the user's Gmail web interface. We do this by creating an empty label/folder nested under the folder/label "myriad". As we can never rely on the "myriad" parent label already being present in a user's account (e.g. because they accidentally deleted it), we check for this first and create the label if necessary. The CampaignLabelCreator receives a new job whenever a new campaign is created or updated with the checkbox for Label Syncing ticked.

Additional points of interest

Email sending

Compared with the fetching process, sending emails is relatively straightforward. When creating a new email, a new job is enqueued in the MessageSender. Only emails that are of type *OutgoingCreatedEmail* and have delivery_status 1 (= created) will be sent. After sending, the status will be changed to *sent (= 3)*. If sending fails for three times, the status will be changed to *sending_failed (= 4)*. If a Delivery Failure Notification for this email is detected, its status will be changed to *delivery_failed (= 5)*.
Our major goal during development was to **never send email twice**. When manually

manipulating the database, please be aware that simply resetting an *OutgoingCreatedEmail's* delivery_status to 1 (= *created*) will cause it to be sent out again - **please use caution when changing delivery statuses**!

The importance of the Allmail folder

As we want to fetch not only emails received by the user, but also emails sent out by them, we need to monitor the *All Mail* folder in their Gmail account (instead of just the *Inbox*). The *All Mail* folder contains all emails of a user's Gmail account, including sent emails and drafts.
It is possible to hide certain folders to IMAP requests (Gear icon > Settings > Labels). If a user's All Mail folder is not visible to Myriad via IMAP, fetching will fail. **Important:** Do not simply switch to querying that user's *Inbox* (or any other folder) instead. This will screw up our database as UIDs are not consistent across folders. We would both start missing emails and constantly try to refetch others we already fetched.
Note: As the All Mail folder may have different names in different locales, we identify it by its universal flag AllMail. This should work for now, yet if you need more information this thread might be helpful.

Users with multiple email addresses

When Myriad fetches emails, it compares the new emails' from addresses with the user's account email address in order to determine whether the email is incoming or outgoing. As some users may have set up one or more alias email addresses with their Gmail account, emails sent from those alias addresses would be falsely categorized as incoming. In order to prevent this, users may add those addresses to their Myriad account. They will then be considered along with the account email address when fetching emails.

Gmail quirks - fun obstacles to mix things up

Sometimes, Gmail does interesting things.

IMAP requests behavior

It is possible to query for an email's header message id and references with analogous syntax, yet it is only possible to search for emails by header message id, not by references.
This is one of what we remember as multiple inconsistencies. We can't recall specifics on the others - just be warned that they do seem to exist, so if stuff doesn't work even after triple-checking your syntax, it might not be you who is at fault. Of course, we might have also

erred with a few things listed in this section to due to our limited knowledge of the technologies at work. If so, please feel free to correct our statements here.

Labels

Labels function like folders in Gmail, so you would think labels are assigned on a per email basis. And this is actually true - if you set or query them directly via IMAP requests. Yet, in the Gmail web interface, you can only apply labels to whole conversations. When you do this, all emails in this conversation will receive the label. Once new emails are added to this conversation, they will **not** automatically the same label, yet it will still look in the interface as if the whole conversation is labeled. Here is a blog article discussing this behavior (it's from 2008 but it seems to still be relevant).

Miscellaneous

Delivery Status Notifications

Gmail does not seem to conform to all standards regarding Delivery Status Notifications (see https://github.com/mikel/mail/issues/103). We extended the Mail class to cope with this (see config/initializers/mail_extension.rb).

Thread size limit

Gmail threads are limited to contain 100 emails each, causing Gmail to create a new conversation for the 101st email. The new conversation has a different thread id than the previous one, which means that Myriad will be unaware of it until it picks up on the first email in this new conversation. If, for instance, the user adds the 101st email by composing and sending an email from his Gmail account (-> OutgoingFetchedEmail), this email will initially be invisible to Myriad. Myriad will find this conversation once there is an email added to it that is in reply to a Myriad identifier or when the user labels the conversation manually.

Net::IMAP and Gmail Imap extensions

For Ruby, the Net::IMAP class provides IMAP client functionality. All methods used by the current Myriad implementation are wrapped by (Myriad's) Gmail class.
Gmail extends the IMAP protocol with some specific features. Among other things, this enables us to perform advanced searches (as available in the web interface) also via IMAP. Also, it allows us to query for Gmail-specific identifiers, such as thread ID and Gmail message ID).
In order to make Net::IMAP ready to work with Gmail extensions, we copied code from

GmailValet that monkeypatches the Net::Imap class accordingly (see *lib/imap_extensions.rb)*.

Rake maintenance tasks

Fetching emails

rake gmail:fetch_all_users

Fetches emails for all users. This task is currently run once a day for maintenance (as specified in *config/schedule.rb*).

Fetching UIDs/THRIDs

rake fetch:refetch_missing_ids

For every user, tries to refetch any UIDs/THRIDs that are missing in our database.

rake fetch:refetch_recent_missing_ids

Same as refetch_missing_ids but limited to the past two weeks for performance reasons.
If we fail to retrieve an email's UID and/or THRID right after we send it for three times, we give up. As this is bound to happen every once in a while, this task is currently run once a day for maintenance (as specified in *config/schedule.rb*).

rake fetch:refetch_missing_ids_for_user user

Same as refetch_missing_ids but only for one specific user. Takes a user id as a parameter.

rake fetch:refetch_all_ids

Deletes all UIDs/THRIDs for all users and tries to fetch them again.
WARNING: Running this task can take a long time and lead to loss of data (in case we don't manage to refetch all ids that were deleted, e.g. because the user deleted the emails in their inbox). Usually we should not need to run this at all. It is basically an emergency solution if we for some reason fetched wrong ids.

rake fetching:refetch_all_ids_for_user

Same as refetch_all_ids but only for one specific user. The same warning applies. Takes a user id as a parameter.

Future improvements

There is a wide range of potential improvements that can be made to the current fetching implementation. As stated, we feel that fetching is relatively reliable barring some edge cases, so none of these improvements are urgent. Yet, if you need some inspiration, find below some things we would have liked to do but did not have the time to:

- Better exception handling
- Introduce tests for message fetching (if feasible)?
- Use Gmail message-identifiers instead of UIDs as they are unique across folders? Drawback: This solution would only work with Gmail.
- The fetching of UID/THRID for sent emails (see worker IdFetcher) in its current form was introduced towards the end of our stay. It still does not seem reliable, especially when many emails are sent out at a time. Any ids we miss will be refetched once a day, but in order to ensure immediate fetching of incoming replies (which is partially based on THRIDs), the retrieval of ids should be made more reliable.

Development resources and debugging tips

General remarks about debugging

When a certain email is not fetched as expected, usually the reason is either that we are unable to identify the email on the Gmail server (i.e. the email's UID is not returned by the search query we send to Gmail) or that something goes wrong while importing the email. A good tool to investigate the former is to query Gmail "manually" via the command line (see "IMAP via command line" below). For the latter it is useful to look at the fetching log (*log/message_fetcher.log*) and check where things went wrong (in most cases, the email's header message id will be documented there). To investigate this email in your Gmail interface, simply use the search command "rfc822msgid:<header_message_id> right from your Gmail inbox. This command is part of Gmail's Advanced Search.
Besides logs and manual IMAP querying, a third powerful option for debugging is the rails console. Some commands are described in the next section. For executing code step-by-step or inspecting variables during runtime, insert "binding.pry" almost anywhere in the code (see gem *pry-debugger*). As soon as execution reaches this line, execution will pause and wait for your input. Your are now able to examine variables or run other commands right from the console.

Useful rails console debugging and maintenance commands

First start up the console by calling "rails c" from inside your Myriad folder structure.

Fetch emails "manually" for a user (id = 1) and the default time limit, which is currently set to the past two weeks:

```
MessageFetcher.perform 1
```

Override time limit:

```
MessageFetcher.perform 1, 1.year
```

Create an IMAP session in the rails console directly using Net::IMAP and look at the folder structure:

```
# Manually create imap session
user = User.find(1)
imap = Net::IMAP.new 'imap.gmail.com', '993', true, certs = nil, verify =
false
imap.authenticate 'XOAUTH2', user.email_address, user.access_token
# Get whole folder tree
puts imap.xlist("", "*").map(&:name)
# Get only first sublevel of folders
puts imap.xlist("", "*/%").map(&:name)
```

Trigger individual fetch modes for user with id = 1:

```
# Create fetcher and gmail object (which handle imap communication):
user = User.find(1)
fetcher = Fetcher.new(1, 1.year)
gmail = fetcher.instance_variable_get '@gmail'
# Connect and choose folder
gmail.connect(user)
gmail.examine_folder("special-issue")
# Trigger fetching modes
fetcher.fetch_mode_reply
fetcher.fetch_mode_conversation
fetcher.fetch_mode_label
```

IMAP via command line

For debugging purposes, it has proven very useful for us to examine emails and folders by issuing IMAP queries via the command line. This will also give you a better grasp of what our Gmail class is doing behind the scenes in conjunction with the Net::IMAP class.
Note: As Net::IMAP mostly wraps actual IMAP commands (among them the ones demonstrated below), its documentation can also be helpful when looking for information on IMAP commands themselves.

Connect to the Gmail IMAP server via OpenSSL (on Ubuntu):

```
openssl s_client -connect 'imap.gmail.com:993' -crlf
```

Setting up the connection

As communication with the IMAP server is not necessarily synchronous, we must prefix each of our requests with a command tag. The server prefixes its response with the same command tag we used in our request. It doesn't matter what you put for your tags, we will just use ascending integers in our examples. Capitalization of the commands is not necessary, we only do this for better readability. The query syntax uses reverse polish notation.

In order to do anything, we need to login into our account first:

```
1 LOGIN christian.ikas@gmail.com <password>
```

The server response should look something like this:

```
* CAPABILITY IMAP4rev1 UNSELECT IDLE NAMESPACE QUOTA ID XLIST CHILDREN
X-GM-EXT-1 UIDPLUS COMPRESS=DEFLATE ENABLE MOVE ESEARCH
1 OK christian.ikas@gmail.com Christian Ikas authenticated (Success)
```

We can now list the available mailboxes (aka folders).

```
2 XLIST "" "*"
* XLIST (\HasNoChildren \Inbox) "/" "Inbox"
* XLIST (\Noselect \HasChildren) "/" "[Gmail]"
* XLIST (\HasChildren \HasNoChildren \AllMail) "/" "[Gmail]/All Mail"
* XLIST (\HasChildren \HasNoChildren \Important) "/" "[Gmail]/Important"
* XLIST (\HasChildren \HasNoChildren \Sent) "/" "[Gmail]/Sent Mail"
* XLIST (\HasNoChildren \Spam) "/" "[Gmail]/Spam"
* XLIST (\HasChildren \HasNoChildren \Starred) "/" "[Gmail]/Starred"
* XLIST (\HasChildren \HasNoChildren \Trash) "/" "[Gmail]/Trash"
* XLIST (\HasChildren) "/" "myriad"
* XLIST (\HasNoChildren) "/" "myriad/fetch-forwarded-email-test"
* XLIST (\HasNoChildren) "/" "myriad/label-test"
* XLIST (\HasNoChildren) "/" "myriad/spreadsheet-test-campaign"
* XLIST (\HasNoChildren) "/" "myriad_ignore"
2 OK Success
```

Note: Labels in your Gmail function as folders, which means that creating a new label in Gmail will create a new mailbox/folder that will be visible via this command.

As users might have thousands of folders/labels in their account, it is generally recommended to limit the query to the first level of subfolders as follows:

```
3 XLIST "" "*/%"
```

If we want to get everything nested below the folder "myriad", we do the following (here we don't expect more than one level of subfolders, so we can pass *"*"*):

```
4 XLIST "myriad" "*"
* XLIST (\HasNoChildren) "/" "myriad/fetch-forwarded-email-test"
* XLIST (\HasNoChildren) "/" "myriad/label-test"
* XLIST (\HasNoChildren) "/" "myriad/spreadsheet-test-campaign"
4 OK Success
```

In order to search for or retrieve emails, we will have to let the server know which folder we want to operate on. We choose to examine the Allmail folder (as it is a predefined Gmail folder it is nested below "Gmail"). Note that all our following searches are scoped on the emails in the folder we choose here.

```
5 EXAMINE "[Gmail]/All Mail"
* FLAGS (\Answered \Flagged \Draft \Deleted \Seen $Forwarded $MDNSent)
* OK [PERMANENTFLAGS ()] Flags permitted.
* OK [UIDVALIDITY 11] UIDs valid.
* 905 EXISTS
* 0 RECENT
* OK [UIDNEXT 1059] Predicted next UID.
5 OK [READ-ONLY] [Gmail]/All Mail selected. (Success)
```

Note: "EXAMINE" gives us read-only access to a mailbox. For read-write access use the "SELECT" command instead. Please make sure to use SELECT only when necessary (in the current version of the code that is only when we apply labels to an email).

Searching for emails

The SEARCH command allows us to search for email in a mailbox by a variety of criteria. The server responds with a list of sequence numbers. As we usually are not interested in sequence numbers, we use the UID SEARCH command instead, which does the same but returns a list of UIDs.

To search for an email with a certain header message id:

```
6 UID SEARCH HEADER message-ID 57d338b3-223e-4b4c-961d-83e553a0d0b2@email.android.com
* SEARCH 993
6 OK SEARCH completed (Success)
```

Note: The Gmail IMAP Extension "X-GM-RAW" enables us to use the Advanced Search available in the Gmail interface for queries to the IMAP server. This allows for the alternative syntax:

```
7 UID SEARCH X-GM-RAW rfc822msgid:57d338b3-223e-4b4c-961d-83e553a0d0b2@email.android.com
* SEARCH 993
7 OK SEARCH completed (Success)
```

Now to find all emails that are *in reply to* this header message id:

```
8 UID SEARCH HEADER in-reply-to 57d338b3-223e-4b4c-961d-83e553a0d0b2@email.android.com
* SEARCH 1059
8 OK SEARCH completed (Success)
```

We can also search for any emails that were sent or received since a certain date:

```
8 UID SEARCH SINCE 25-apr-2013
* SEARCH 1051 1052 1053 1054 1055 1056 1057 1058 1059
8 OK SEARCH completed (Success)
```

Or emails we received from a certain address:

```
9 UID SEARCH FROM mailmergerer@gmail.com
* SEARCH 839 840
9 OK SEARCH completed (Success)
```

To combine these (the AND operator is implicit from concatenation):

```
10 UID SEARCH FROM christian.ikas@gmail.com SINCE 22-apr-2013
* SEARCH 953 954 955 956 957 958 960 962 964 966 967 968 969 970 971 973 989
992 995 997 1059
10 OK SEARCH completed (Success)
```

Thanks to the Gmail IMAP Extensions, we can also search for emails with a certain label, e.g. "findme":

```
11 UID SEARCH X-GM-RAW label:findme
* SEARCH 1060 1061 1064
11 OK SEARCH completed (Success)
```

Note: You might have to issue the EXAMINE command again in order to make labels visible that you just assigned in your Gmail interface.

This will find emails that have both the label "findme" and the label "findmetoo":

```
12 UID SEARCH X-GM-RAW label:findme X-GM-RAW label:findmetoo
* SEARCH 1064
12 OK SEARCH completed (Success)
```

This will find all emails with the label "findme" but without the label "ignoreme"

```
13 UID SEARCH X-GM-RAW label:findme NOT X-GM-RAW label:ignoreme
* SEARCH 1060 1064
13 OK SEARCH completed (Success)
```

This will find all emails with either "findme" or "findmetoo":

```
14 UID SEARCH OR X-GM-RAW label:findme X-GM-RAW label:findmetoo
* SEARCH 1060 1061 1062 1063 1064
14 OK SEARCH completed (Success)
```

Putting things together, in order to get all emails that were sent from *ikas@stanford.edu* since *Apr 22, 2013*, and that do not have the labels *"boring"*, *"irrelevant"* or *"spam"*, we do the following:

```
15 UID SEARCH FROM ikas@stanford.edu SINCE 22-apr-2013 NOT OR X-GM-RAW
label:boring OR X-GM-RAW label:irrelevant X-GM-RAW label:ignoreme
* SEARCH 959 961 963 965 972 994 996 1054 1055 1060 1061 1062 1063 1064 1068
15 OK SEARCH completed (Success)
```

Fetching emails

For fetching the actual emails, we use the FETCH (or in our case: UID FETCH) command. As parameters we need to pass the UID of the email and what kind of data we want to fetch. Fetching a complete email (with UID 959):

```
16 uid fetch 959 rfc822
* 806 FETCH (UID 959 RFC822 {1790}
Delivered-To: christian.ikas@gmail.com
Received: by 10.52.17.136 with SMTP id o8csp71712vdd;
        Mon, 22 Apr 2013 16:15:40 -0700 (PDT)
[...]
16 OK Success
```

Fetching only the header...

```
17 UID FETCH 959 BODY[HEADER]
* 806 FETCH (UID 959 BODY[HEADER] {1748}
Delivered-To: christian.ikas@gmail.com
[...]
Message-ID: <5175C49A.40807@stanford.edu>
Date: Mon, 22 Apr 2013 16:15:38 -0700
From: Christian Ikas <ikas@stanford.edu>
[...]
17 OK Success
```

Fetching a certain attribute, e.g. the header message id:

```
18 UID FETCH 959 BODY[HEADER.FIELDS (MESSAGE-ID)]
* 806 FETCH (UID 959 BODY[HEADER.FIELDS (MESSAGE-ID)] {45}
Message-ID: <5175C49A.40807@stanford.edu>
)
18 OK Success
```

Fetching the Gmail message id using Gmail IMAP extensions:

```
19 UID FETCH 959 X-GM-MSGID
* 806 FETCH (X-GM-MSGID 1433060025622599424 UID 959)
19 OK Success
```

Fetching the Gmail thread id:

```
20 UID FETCH 959 X-GM-THRID
* 806 FETCH (X-GM-THRID 1433060025622599424 UID 959)
20 OK Success
```

Fetching an email's labels:

```
21 uid fetch 959 x-gm-labels
* 806 FETCH (X-GM-LABELS ("\\Inbox" "\\Important" myriad/55th-campaign) UID
959)
21 OK Success
```

Creating, querying and assigning labels

Create an empty label/folder *myriad*

```
22 create myriad
22 OK Success
```

Create an empty campaign label *test-campaign* nested below the label *myriad*

```
23 create myriad/test-campaign
23 OK Success
```

Delete a label

```
24 delete myriad/test-campaign
24 OK Success
```

Assign a label to an emaiL (must be in read-write mode → SELECT):

```
25 uid store 959 +x-gm-labels myriad/test-campaign
* 806 FETCH (X-GM-LABELS ("\\Inbox" "\\Important" myriad/test-campaign
myriad/55th-campaign) UID 959)
25 OK Success
```

Remove a label from an email (must be in read-write mode → SELECT):

```
26 uid store 959 -x-gm-labels myriad/test-campaign
* 806 FETCH (X-GM-LABELS ("\\Inbox" "\\Important" myriad/55th-campaign) UID
959)
26 OK Success
```

Don't forget to

```
27 LOGOUT
* BYE LOGOUT Requested
27 OK 73 good day (Success)
```

Logs

In the folder *myriad/log*, there are several logs documenting the actions in the message fetching, ID fetching and label setting processes. Especially in the error prone message fetching process, they might be helpful for monitoring and debugging purposes (you might want to adjust them to your tastes).

GmailValet

As GmailValet was heavily concerned with IMAP-based communication with Gmail, we were able to copy or adapt some code or at least use it as source of inspiration. Consider using it as a

resource - there might be some useful code hidden in there somewhere that we either missed or did not have use for at the time, which will save you some time and trouble.

RFCs and other online resources

Various links that came in handy for us:

Standards/RFCs:

RFC822: Standard for the Format of ARPA Internet Text Messages
https://tools.ietf.org/html/rfc822
RFC2822: Internet Message Format (obsoleted by RFC5322)
https://tools.ietf.org/html/rfc2822
RFC5322: Internet Message Format
https://tools.ietf.org/html/rfc5322
RFC3501: Internet Message Access Protocol - Version 4rev1
http://tools.ietf.org/html/rfc3501

Gmail doc

Overview of standard Gmail folders
http://support.google.com/mail/answer/82367?hl=en
Advanced search from the Gmail interface
http://support.google.com/mail/answer/7190?hl=en
Gmail IMAP extensions
https://developers.google.com/google-apps/gmail/imap_extensions

Gmail discussion

Stackoverflow about localized IMAP folders
http://stackoverflow.com/questions/2185391/localized-gmail-imap-folders
Blog article discussing weird Gmail labeling behavior
http://www.jamesmurty.com/2008/03/13/gmail-labelling-search-bug/

Gems

Regarding Gmail bounce handling by the Mail gem:
https://github.com/mikel/mail/issues/103

Other

Imap::Net
http://www.ruby-doc.org/stdlib-2.0/libdoc/net/imap/rdoc/Net/IMAP.html

More Helpful Tacit Knowledge

Magical Ruby on Rails

Rails is awesome. It works like magic. Rails is terrible. It does so many things without you telling it to. *Don't give up*. Learn about observers and callbacks. They feel like magic at first, but allow us to do abstract *concerns*. For example we observe all objects and destroy any notifications attached to them when they are destroyed. You don't have to worry about that when you implement a new class, but it also isn't visible to you if you don't know about it. We'll try our best to tell you all there is to know. **But if something happens and you don't know why, check the observers and the class for callbacks.**

On "proper" database schemata

We gave up trying to force "proper" db schemata into Rails. Yes, an email attachment shouldn't have it's own primary key, it should just use Email's. But we couldn't get ActiveRecord to behave nicely in those situations. Same with composite primary keys, even though there exists a gem for this. So we just accepted that Rails has an ID for every resource, using indexes with unique requirements and validates-associated-validations to create some sanity checks at least.
For now we don't think it's worth fighting Rails' defaults, strange as they may feel.

On mysql

Everything points to PostGreSQL being the better, more modern database. We just didn't have the courage to migrate. Feel free to do so. No good reason to stay but laziness. :D

Class diagram

A human created class diagram is available at the project's github repo at the `/doc` folder.

Generate class diagrams

We found a nifty gem called railroady that can generate class diagrams automatically. We didn't find them super useful once you know the codebase, but they might prove useful to you.
To generate them, run `rake diagram:all`. You have to have graphviz installed. Mac users

will want to use `brew install graphviz`, ubuntu users can use `sudo apt-get install graphviz`.

Generate db schema annotations

Run `annotate` to generate helpful annotations that put a resource's schema information into their respective source files. That's super handy after creating a new resource, when you have set up some db constraints and want to back them with Rails-side validations, for example.

The good and bad about admin user accounts

Developing as an admin user is nice, because you can use the notification system to be informed about exceptions as they happen even on background workers.
However, developers often didn't catch bugs that happened only for non admin users. Our custom authenticate method in CanCan's ability.rb now makes this less likely, but it's still worth it to test a new resource/controller with a non admin user.

rake db:validate

We built a custom rake task that goes over the entire database and validates every single object. When it's finished, it displays a report, and offers to delete invalid records. This takes minutes, but can be super useful if you later discover you hadn't introduced a specific constraint, or if you worked directly on the db. We usually ran it before a db change, made sure there were no invalid records, made our changes and ran it again.

cap deploy:quick & deploy:full

When we introduced capistrano into our deployment process, we weren't ready to switch to a fully automated capistrano deployment setup, thus writing entirely custom deployment scripts. You currently can't use `cap deploy` alone. This is due to historic reasons, so feel free to switch the project to a more standardized capistrano deploy method in the future. This might be a good place to look after you've learned the basics.
The configuration is found in `/config/deploy.rb` and `config/deploy/[ENV].rb`

`cap deploy:quick` deploys code changes and restarts workers and Rails. It's awesome for hotfixes, as it's quick and doesn't do dangerous stuff.

`cap deploy:full` does a full commit, including bundling gems, asset precompilation, db migration and crontab changes. It usually takes minutes, though.

rake db:backup

Does a backup, *overwriting the last hourly backup*.

rake resque:start_workers, rake resque:stop_workers

Manually starts or stops workers. We used this mostly during times when our monit config was not working for mysterious reasons and during development to restart workers, so they picked up code changes.

rake idle

This one's a beast. It *IMAP IDLE*s on all user's Gmail accounts to enqueue MessageFetching when a new email arrives. Adopted from GmailValet, it's a bit sketchy at times… once it's running, it works, though. Feel free to refactor the heck out of this if you're up for a challenge. On production, it should be monit'd.

monit

Monit is an awesome tool that you configure via a DSL that ensures all parts of your appliction keep running. The most common way for monit to monitor an application is that this application writes its Unix process ID into a specific file. This way monit can check if this process is still alive. Big applications like nginx or mysql already do this for us. For things like workers or the above mentioned rake idle task we need a custom solution. Imho it would be best to use a *deamonizer* wrapper appor script, that does the starting and PID-into-a-file-writing generically. I wasted a few hours on several approaches and couldn't get this to work, unfortunately.
So now our rake idle task writes it's own PID file and is monit'd. Semi-awesome.

Project folder structure

/app
- **/assets**
  - **/images**
  - **/javascripts**
    - **/application**
    - **/plugins**
  - **/stylesheets**

/application
/plugins
/controllers
/helpers
/inputs
/mailers
/mixins
/models
/email
/notification
/redactor_rails
/networking
/observers
/uploaders
/validators
/views
/workers
/art
/config
/custom
gmail.yml
/deploy
/environments
/initializers
/locales
/redis
application.rb
application.yml
database.yml
environment.rb
monit.conf
routes.rb
/db
/migrate
schema.rb
/doc
/lib
/log
/public
/script
/test

/vendor

Our Version of “Best Practices”

> Try not. Do, or do not. There is no try.
>
> — Yoda, *The Empire Strikes Back*

Core Extensions

It’s worth reading through `core_extensions.rb` in one go before looking at other code too closely. We came up with an unfortunate big number of small methods we’d like in our Ruby core classes, especially `String`. Here’re the non-self explanatory highlights:

String

`remove_all_whitespace`

Just what it says. A more semantic version of `gsub /\s+/, ''`

`similar_to?`

A custom equivalence class on strings, which are similar if only numbers and downcased letters are considered. We use this to match placeholders with spreadsheet column headers, for example. Can match mistakes like “first_name” <> “First Name”, i.e:

```
“first_name”.similar_to? “First Name” == true.
```

`ellipsisize`

Takes a subset of a string and adds an ellipsis where it cut it of. Search project for usage examples; e.g. we use it when only displaying the first line of a collapsed email.

`first_name/last_name`

A semantic name for first word/last word of a string. Put here to unify our efforts to extract parts of names from all kinds of strings. Is obviously limited in cultural applicability, as there are lots of cultures with more or less than 2 name components.

Hash

`deep_find`

Instead of hierarchically accessing a value in a Hash it lets you access it flatly, at the risk of only finding the first of several similar keys. We used it to extract `password` and `username` keys

from highly hierarchical Hashes.
Avoid usage if possible; we simply experienced Google's API changing ever so slightly by renaming a parent Hash of the one we were actually interested in, giving us nil exceptions.

Array

`uniq?`

Why this is not in Ruby or even Rails is beyond my understanding. Since almost nobody seems to use `Set`s, this is an OK substitute.

`contains_duplicates?`

Alias of `not uniq?`

`fold`

Usage was later removed for easier understandable code.
But if you like functional concepts, have your fold, foldl and foldr.
Also, do you know this?
Also, would you like to get a coffee sometime?

File

`unique_filepath`

Use this when saving files in a directory where there could already be a file with the same name. If that's the case, this function will give you a new filepath with a numeric suffix large enough to make the filepath unique.
So, you're saving test.pdf in `tests/`, and it already contains a file called `test.pdf`.
You give this function `tests/test.pdf` and get back `tests/test-1.pdf`.

ResqueWorker -> AbstractWorker

We've built an AbstractWorker Superclass for our Resque Workers. Why?
Basically I think Resque is awesome, but very, very basic.
When faced with the task to build a DRY function to start worker threads for all our workers, I found no simple way. So now, AbstractWorker takes care of generating queue names form the Class names of it's subclasses, and I can use the result of the `descendants` method (returns all subclasses) to iterate over and start a worker thread with the correct queue. It also allowed us to extract the ResqueDirector plugin, which starts those workers in our Development

environment.

Coding Style

Method bodys should be 5 +/- 2 lines long. Yeah, right. Actually, yeah, right! take a look at the spreadsheet class for inspiration. But obviously, it's hard to stick to this in an environment with more than one coder. Oh well.

Guidelines we wish we had followed all along

- Lean Controllers, almost no logic.
- Fat Models, but seperated into aspects.
- Meta-Model Aspects go into Observers.

Setup

A wish

*Hey, you! Yes, you! You have the once in a project-lifetime chance to do the right thing® and setup a development and production environment using **vagrant** and **chef**. It'd be awesome if you read about those technologies and banned inconsistent development/production environments once and for all.*

On Development Machines

Ubuntu

This, almost definitey, won't work for you as is. We have compiled all the steps Chris did, but things might have changed. Use this more as a guideline if you're missing something and don't know why.

Install Ruby & Git

```
sudo apt-get install build-essential git-core curl
sudo apt-get install zlib1g-dev libreadline-dev libssl-dev
libxml2-dev
```

Install RVM

```
\curl -L https://get.rvm.io | bash -s stable --ruby
```

```
source ~/.rvm/scripts/rvm
```

Install RVM requirements

```
apt-get install build-essential openssl libreadline6 libreadline6-dev
curl git-core zlib1g zlib1g-dev libssl-dev libyaml-dev libsqlite3-dev
sqlite3 libxml2-dev libxslt-dev autoconf libc6-dev ncurses-dev
automake libtool bison subversion pkg-config
```

Set up correct Ruby version in RVM (feel free to try newer versions)

```
rvm install 1.9.3
rvm use 1.9.3 --default
rvm rubygems current
```

Install rails

```
gem install rails
gem install bundler
sudo apt-get install rails
```

Install JavaScript Runtime

```
sudo apt-get install libpq-dev python-software-properties
sudo add-apt-repository ppa:chris-lea/node.js
sudo apt-get update
sudo apt-get install nodejs npm
sudo apt-get install redis-server
```

Deploy to local server

```
git clone https://github.com/taskpot/myriad.git
bundle install
rake db:create:all
rake db:migrate
rails s
```

Mac OS X

As a general remark I found it rewarding to set up most things in the way you would on linux,

instead of using GUI tools. You'll need it for the production server anyway. :/

It follows that you just follow the Ubuntu steps; skipping over all apt-get calls.

I'll just quickly outline setting up a generic unix-y development environment on your mac:

1. *Download and install XCode from the app store, and install developer tools.*
2. *Install Homebrew, a package manager*

   ```
   ruby -e "$(curl -fsSL https://raw.github.com/mxcl/homebrew/go)"
   ```
3. Set up defaults as per http://mths.be/dotfiles
4. Install mysql: brew install mysql
5. Install redis: brew install redis

My opinions on tools

If you end up using a better/newer alternative I'm super interested to hear about that!

Text editor

Should support your workflow with shortcuts.

Gui editors: TextMate, Sublime, MacVim

We used Chocolat (not free, not necessarily worth the money) and Sublime.

CLI editors: vim, emacs

Terminal

I'd really recommend iTerm if for nothing else then for its fullscreen and splitscreen modes.

Those are useful for monitoring log files while working in the rails console, etc.

db browser

Sequel is hands-down one of the nicest developer tools I know.

git browser

Source tree is handy and comes with git flow built in.

Lacks a proper per-file history view; at least I couldn't find it.

IDE

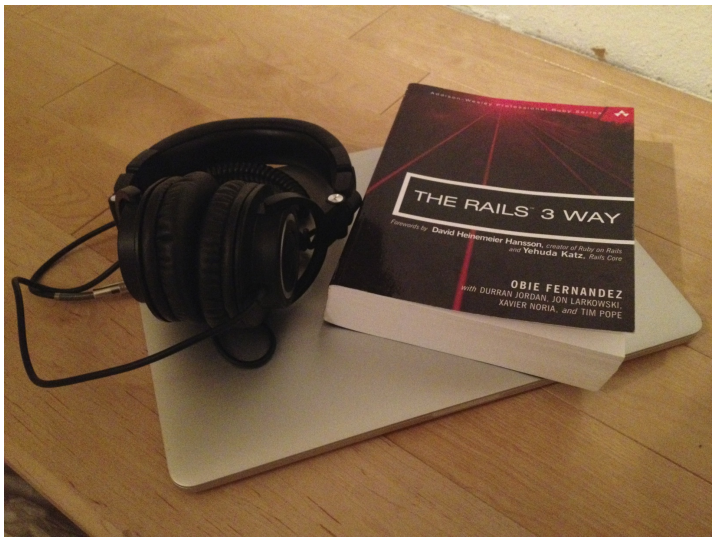
Nicolas/Michael might be able to offer you a RubyMine license.

I'm usually a big fan of IDEs, but not of this one. I'd suggest you give it a try, though!

Music

You're in the US; get a free Spotify subscription for a few months! :D Bach's Brandenburg

Concertos are awesome for coding if you can stand the festivity, or try Satie's works otherwise. You'll have to use headphones, though.



My workplace in the first week. :-)

Late-night work

Try f.lux for the evening; Nocturne set to red tinted and inverted at night.
Won't screw up your circadian rhythm as much. Also, they freely sell Melatonin here! Helped me get back to a healthy sleep schedule a few times. Just don't take it regularly.
There's almost no food here at night, so either be prepared, or just drink a lot of water. Also, California gets *really* cold after midnight.

Much love for reading this far.
Good luck.
L/C/B