Planned Deletion Emails

Members:
CANIN Corentin, MONTEILLER Joshua & WAGNER Samy

Tutor:
PÉRIN Michaël

Department:
INFO4 - Polytech Grenoble - UGA

UGA Université Grenoble Alpes

POLYTECH

Table of content

Table of content 2

Introduction - The problem 3

Ideas of solution 4

Absolute Date 4

Relative Date 4

Other actions 5

Design decisions 5

Thunderbird: our work base 6

What are Thunderbird extensions 6

What Thunderbird allows us to do 7

Our work and our choices 8

From the HTML header… 8

… to the attachment in the email 9

The user interface 11

The menu popup 12

The composer popup 12

Going further - what was done, what can be improved 13

Conclusion 15

Introduction - The problem

Every day, we receive a great number of emails, from various sources, from our parents to our friends, from unwanted spams to professional emails... Inboxes are constantly filled with messages and the quantity always keeps increasing. The only solution nowadays is to manually sort all our emails and put in the trash everything that is unwanted or useless, or emails about past events or things that are no longer needed and have become outdated. It can be a harshful task to sort them when we sometimes receive more than 10 emails per day.

Not only is it annoying for the user but it is also harmful for the environment. Modern emails are stored in servers, often in huge datacenters, consuming enormous amounts of energy to keep everything running and cool the machines down. In many countries, most of the electricity used to power these servers comes from fossil fuels, which produce greenhouse gases that actively harm our planet. A mere email sent last month about a meeting that happened an hour later is a threat to our environment, as well as a financial abyss and a waste of energy that could power houses or critical structures.

As emails are still useful in many situations and are the preferred way to communicate in various contexts such as schools or companies, we can't just stop using them. A better solution would be to make the task of cleaning our inbox less daunting by automating it. To do so, a solution to create deletion conditions (DC) on emails in order to delete them automatically was imagined.

Ideas of solution

In order to automatically delete emails and save resources, a potential solution was drafted by Michaël PÉRIN and Samuel CONJARD. This section will explain their initial proposal, which we used as a basis when developing a solution. The main idea of their work of research is to put somewhere in the email a deletion condition. These conditions could be in an email header field, or the subject of the message, and could be stored inside tags that would indicate where the conditions are and may eventually allow hiding them. They are declined in various categories:

Absolute Date

The idea here is to delete the email on an absolute date. This would prove useful for emails concerning meetings or appointments. Once these events have taken place, there is no need to keep the email in our inbox. The sender could set the deletion condition to the event's date, automatically deleting the email past it.

Example: `Your ticket for the concert 2021/10/01 <DC>2021/10/02</DC>`

Relative Date

Other emails are only useful for a very short period of time: small notes, short answers. For those cases, it would be useful to have the emails deleted after a small timespan after either sending or reading them. To do this, an additional syntax to add inside the DC tags was proposed to tell which action would trigger the timer.

This concept can be expanded further by introducing variables to store a date and add a timer to it in a DC condition elsewhere.

Examples:

- `Thanks <DC>@READING</DC>`
- `[MI6] Here is your mission <DC>@READING +1min</DC>`
- `The meeting is in room B20 <DC>@SENDING +1hour</DC>`
- `Your ticket for the concert DATE=2021/10/01 <DC>@DATE +1day</DC>`

It was suggested that `<DC>@SENDING +MAX_DURATION</DC>` could be the default DC.

Other actions

Similarly to relative dates, we can imagine all sorts of actions that would trigger a timer such as saving an attached file, replying or even the author requesting the deletion with a new email.

These DC could eventually be combined with disjunctions of conjunctions in order to have more complex conditions.

Design decisions

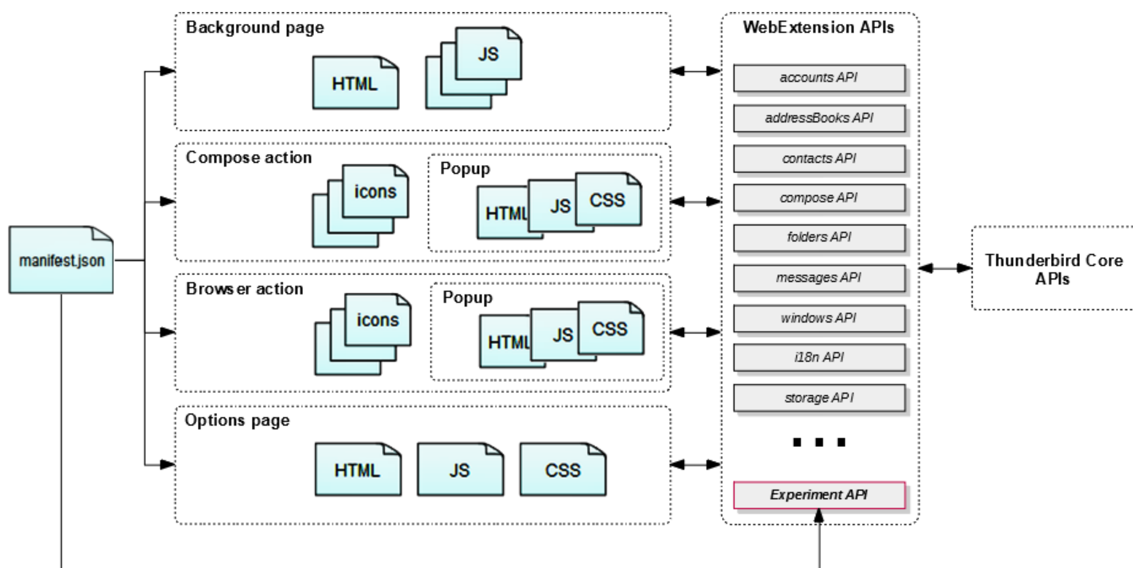
The proposal suggested that a sort of programming language to write deletion conditions could be used to let users fully customise the DC. A field in the email composer would allow these conditions to be edited, but an additional user interface to make things easier and clearly show the condition to the recipient. To implement this idea, it was suggested to build a Thunderbird extension as it is the only mailer to allow such a high level of customization. To a user who doesn’t have the extension, an email with a deletion condition should appear as a normal email, but those who have it must be able not only to have emails with DC automatically deleted and to tell these apart but also to change the DC of received emails and disable automatic deletion for some time or even entirely, for example.

Thunderbird: our work base

It was chosen to try and implement these ideas as a Thunderbird extension. While Thunderbird is not the most widely used mailer software, it is still quite popular and no other mailer offers us all the tools we need to build this tool, although even with this level of customization, some features that we would have wanted were still lacking. With that in mind, it became clear to us that we still had to provide a lot of effort in order to implement the ideas Mr. PÉRIN and Samuel CONJARD proposed.

What are Thunderbird extensions

Similarly to web browsers such as Google Chrome or Mozilla Firefox, you can install add-ons to Mozilla Thunderbird, called MailExtensions, to customise your user experience. They consist of a collection of files similar to those of a typical website, with HTML pages styled with CSS, JS scripts and different types of resources such as pictures. These files are listed in a JSON file named “manifest.json”, that is used as a root for the extension in Thunderbird. Scripts can behave very differently, run on various places and have access to different subsets of the MailExtensions API provided by Mozilla to manipulate emails, folders and other elements that constitute the Thunderbird environment.



Structure of a Thunderbird MailExtension
Source: [Thunderbird]

MailExtensions are actually based on the same technology as many web browsers’ extensions like Firefox: WebExtension. However, Thunderbird supporting

WebExtensions is very recent (2018-2019) and because of this, a lot of features are yet to be implemented in the Thunderbird API.

For instance, even if a user can do it manually, it is impossible to automatically add a custom field to an email via a Thunderbird Extension preventing us from implementing this idea. Generally speaking, a lot of time at the beginning of our project was dedicated to discovering the Thunderbird API and what was made possible by it and what wasn't.

What Thunderbird allows us to do

Because of the MailExtension API's limitations, we were unable to add a new field to the email, which led us to completely drop the proposed syntax altogether. We thought it wasn't that useful when an user-friendly interface was also implemented alongside it.

With this issue came more work: how to store the condition if it isn't in a field? Should it be in the subject, the email's body, or even elsewhere? We came back on this subject multiple times during the project, showing how much of a trouble this question posed us. We also realized while looking at the MailExtension documentation that along with this, many other features, thought of lesser priority, such as deleting an email when an attachment is opened, wouldn't be doable.

That being said, we still had a lot of options to work with: HTML pop-ups for the user interface, LocalStorage to keep track of emails that need to be deleted and when, events that are triggered when an email is received, sent or read…

Our work and our choices

From the HTML header…

In the beginning, we thought a lot about how to add a DC in an email. A key part of our idea was that it must be invisible to the user because the user must not interfere with the DC, it can make it useless if a point, a number or a letter is added or erased from the DC. An invisible deletion condition also means that users won't be disrupted by the sudden appearance of a new element that may lead them to wonder about what it could be and if their Thunderbird is broken. In short, we tried to build the most user-friendly environment, as the add-on might be used by people who are not used to computers.

```
@READING | @SENDING +30min ->

<dc>
  <dc-or>
    <dc-read/>
    <dc-send time="1800"/>
  </dc-or>
</dc>
```

Example of how we thought the idea could have been implemented

We decided to add the DC in the HTML head of each email. Consequently, each email sent by Thunderbird with a DC needed to be in HTML format and the plain text format was forbidden to use, although we realized it was impossible to know what format an email being written uses as the field the API gives us to do that doesn't work.


```
{
  "abs": {
    "2022": {
      "2": {
        "29": {
          "4": [
            {
              "id": "64f9c7b4@gmail.com",
              "time": "30"
            }
          ],
          "8": [
            {
              "id": "99a9b8b4@gmail.com",
              "time": "30"
            }
          ]
        }
      }
    }
  },
  "read": {
    "64f9c7b4@gmail.com": {
      "delay": "1|3|10|5|26",
      "abs": "2022-03-29T04:30:00.000Z"
    },
    "c0700bb2@gmail.com": {
      "delay": "||5|2|"
    }
  }
}
```


In the meantime, we discussed between us how to list all the DC of every email received. A database was needed and we decided to use the JSON format to store the DC in the LocalStorage. The absolute database stored the emails' IDs in a JSON hierarchy, starting with years, then months, days, hours, and lastly the email object with its ID and minutes. This would have allowed us to easily find emails and add them at the right place, or remove whole blocks of expired emails (for example, we could remove the JSON object containing emails from 2021 when checking emails in 2022). Relative deletion times would also be stored in a JSON object, with email IDs as indices. At each index, we would add an object containing the delay, and if the DC also had an

absolute date, the date would be added as well, as finding the absolute condition is easier through the date than through the ID because of the JSON hierarchy. This was needed as relative conditions are converted to absolute conditions when the corresponding email is read.

However, those decisions were not the final ones. In March, at the beginning of the month, we had a halfway presentation where we presented our work. The professors in the jury, Mr. PALIX, Mr. DONSEZ and Mr. RICHARD, showed us that the use of HTML was a false lead. They explained that if we wanted a simpler solution, we should rather use an attachment in the emails. With this indication, we changed our solution.

… to the attachment in the email

We decided to take the jury's feedback into consideration and started working on a different approach to send deletion conditions and rewriting the DC handling code. From then on, the DC would be stored in an email attachment using the lightest possible format. The format is as follows (with an exemple):

2022-03-29T04:30:00.000Z;1|3|10|5|26

In this example, the user wants the email to be deleted on March 29th 2022 at 6:30 or 1 year, 3 months, 10 days, 5 hours and 26 minutes after being read by the recipient. This text is is stored in an attachment named "__dctimedata__" and follows these rules:

- The absolute date is always stored before the relative date.
- The two dates are separated by a semicolon.
- The absolute date follows the [ISO 8601 format](#).
- The numbers for the relative date are separated by a vertical bar and are in this order: year|month|day|hour|minute.
- Before the email is sent, we use a draft version of this format that includes an underscore before a condition that has been unchecked in the popup. In the following example, the relative date is unchecked: 2022-03-29T04:30:00.000Z;_1|3|10|5|26.
- Unchecked elements are completely erased when sending the email, as they are only used to restore the complete popup's state when reopening it or saving the email as a draft.

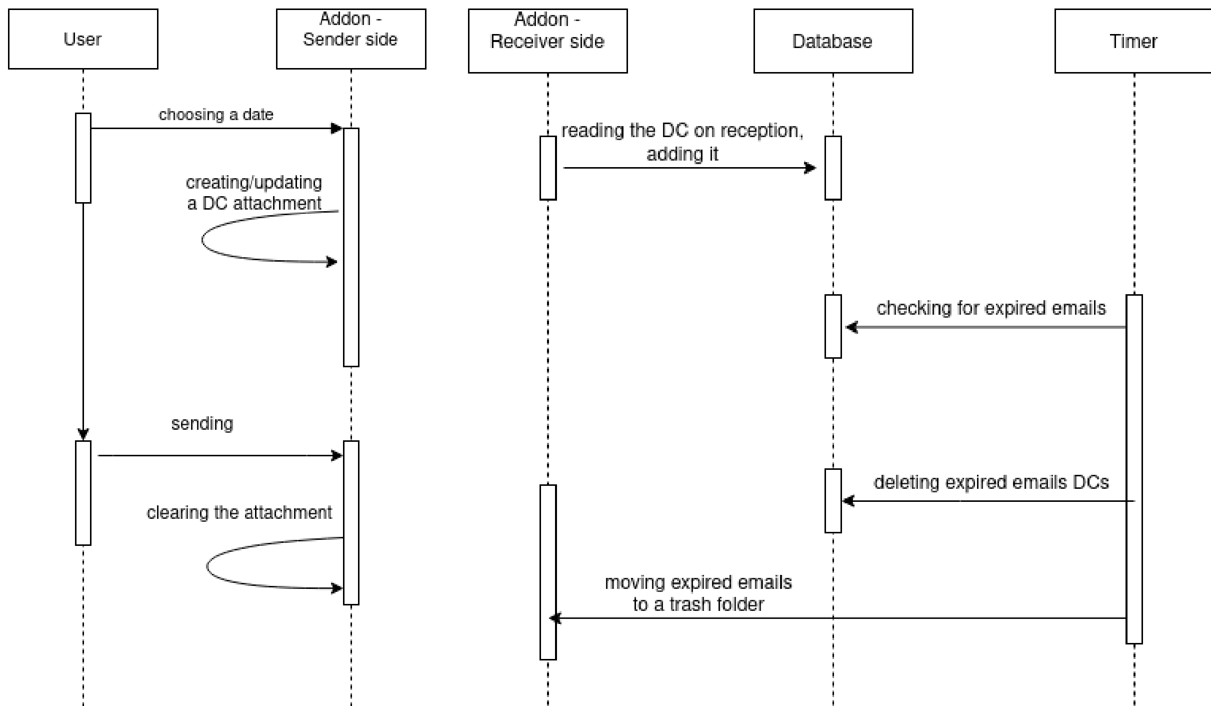
The attachment is lightweight and predictable: the absolute date has a constant length, and the relative time is as large as the numbers used. After cleaning, unused values are completely erased, saving some extra space. If all values are filled, and the relative time is only using one-digit numbers, the attachment weighs only 34 bytes.

```json
{
    "abs": [
        {
            "id": "64f9c7b4@gmail.com",
            "time": "2022-03-29T04:30:00.000Z"
        },
        {
            "id": "99a9b8b4@gmail.com",
            "time": "2022-03-29T08:30:00.000Z"
        }
    ],
    "read": {
        "64f9c7b4@gmail.com": "1|3|10|5|26"
    }
}
```

We also decided to change our database. Absolute dates are stored in an array, sorted in ascending order, of objects containing only the email's ID and deletion time. This means that we can't quickly sort conditions on insertion by hand as we don't have a clear hierarchy anymore, but checking for expired conditions and removing them is faster as we don't have to go through a complex JSON tree. We have to check conditions regularly while adding a condition only happens when an email is received, which means that fast deletion is more interesting than fast insertion. Additionally, depending on Thunderbird's sorting algorithm, insertion speed may not be affected too much. We also changed how we store relative dates, as finding an ID is now as easy as finding a date. We need the email's ID, so we don't have to store the absolute date with it anymore.

This change also largely simplified our code as JS arrays already have methods for inserting, sorting and finding elements on a condition, which means that all the JSON reading could be removed in favor of one-line instructions, and many auxiliary functions were not needed anymore.

Simplifying the database cleared up all the issues we had to implement our deletion routine. The extension's background script only has to regularly check the topmost element in the absolute conditions array and compare it to the current date. If its date is older, the email with the corresponding ID is deleted and the new topmost element is checked, repeating until all expired emails have been deleted. When an email is read by the user, we search its ID in the relative database, and if it exists, it is removed, an absolute date is calculated from the current date and the condition's delay, and is then added as a normal absolute condition. If the email already has an absolute condition, only the earliest date is kept.



Sequence diagram of our addon.
On the left, the actions of the addon on the sender side.
On the right, the actions of the addon on the receiver side.

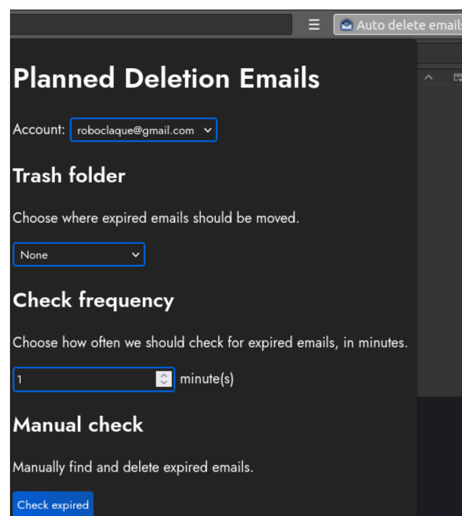
Finally, Mr. PERIN advised us against directly using the trash folder. We initially wanted to send expired emails to the trash as it seemed like the most sensible place to put them. In a discussion with our tutor, we were warned that in the US, we could be sued for deleting an email without the user's consent as it could be detrimental to them. For example, the user may miss a great deal for his company because of a DC in the email, and could say that it's the addon's fault and sue its creators. We took the decision to let the user choose what folder will be used to store the expired emails. With this, we are no longer responsible for losing an important email if the user voluntarily decided to use the trash folder.

The user interface

As stated previously, Thunderbird allows extension developers to add popups at various places of its interface, programmed using standard web languages (HTML, CSS and JS). It automatically creates buttons where needed that will open the corresponding popup when clicked. We decided to use this feature to implement user-friendly interfaces to configure the extensions and add DCs to an email instead of using a complete programming language that would have had to be parsed and would have been difficult to use for the average user.

The menu popup

From this popup on Thunderbird's main inbox tab, users can set their desired trash folder, and how frequently the extension should check for expired emails, as well as triggering a manual check. If users have connected multiple accounts to their mailer, they can select the folder they will use as trash for each account. As Thunderbird runs popup and background scripts in completely separated environments, and the main background script is responsible for checking emails, clicking on the manual check button sends a message to the script through a runtime message API.

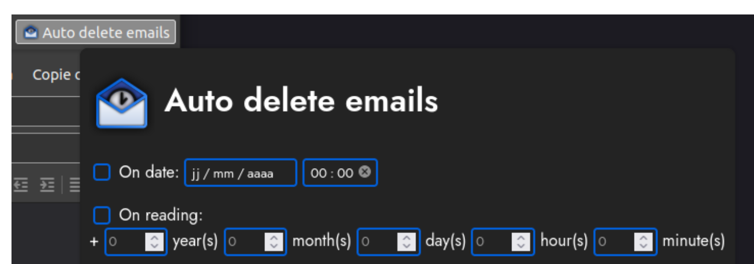


The menu popup

The composer popup

We added a second popup to the composer tabs, which is the interface where users can write new emails. From this popup, they can set the expiration date of the email after which it will be automatically deleted if the recipient also has the extension, as there is no way to trigger the deletion without requiring the extension to be installed everywhere.

Setting the absolute or relative (currently only on-read) date in this popup automatically creates or updates the DC draft attachment described earlier, which stores extra information that will be deleted when sending the email so that a draft can be saved and any value entered in the popup can be restored when reopening it.



The composer popup

Going further - what was done, what can be improved

The extension currently only has the most basic features requested, and more things could be added or improved in the future.

The extension's storage should be changed. We currently use the LocalStorage API to store our conditions database. If a user receives a lot of emails, the database could take up a lot of space, but LocalStorage is limited to only 3 MB. The solution would be to use the IndexedDB API, which gives us access to a JSON-based database that isn't limited in storage space. This API is more difficult to use, so we decided against using it, and we've chosen to build a functional extension first, but a final version of the extension would necessarily have to be moved to IndexedDB.

Users should be able to know if an email is about to be deleted and to cancel the deletion if they want. This could be done by adding tags on emails, and adding another popup on the email reading window that would show how much time is left until the email is deleted, and would give options to postpone the deletion or completely cancel it. Some basic postponing options and cancelling could also be added to a context menu that would allow the user to do these actions without opening the full popup.

Other deletion conditions could be added. Many conditions were suggested in our initial subject: some of them didn't look very useful, such as a delay relative to the sending time (users can use an absolute condition and calculate the delay themselves), but some others could be interesting. These include events like answering the email, receiving another email requesting a previous one to be deleted, or opening an attachment.

When we decided to use the header of HTML-based emails, we wanted to make deletion conditions invisible to the users so that they could not interfere with it outside of the popup and would not see elements appearing seemingly out of nowhere, something impossible to do with attachments. The best solution would be to use an email header field, but these can only be added by the user in Thunderbird's advanced configuration. We could make a suggestion on Mozilla's bug tracker to add an API to create header fields programmatically. The extension would then fill this field with the content of our attachment and read conditions from it, which would further reduce the email's size.

Various improvements could be done on the extension and its popups. Adding an attachment with the same name as our DC attachment or removing the one the

extension created could trigger a confirmation dialog. The original subject also suggested to notify the user when an abnormal condition (such as an absolute date in the past) is used, but we could also do more checks on the attachment file when sending or receiving an email and dynamically add more conditions to the popup, such as defining the minimum date in the selector box to the current date. A default DC option could be added so that any email could be automatically deleted without having to add a condition manually.

Conclusion

This project probably wasn't the most difficult subject out of all those that were proposed to us, but it certainly was one of the most interesting. It required us to think about many design aspects, the number of revisions we did to our conditions database being a clear example of this thinking process throughout the project. Finding the most efficient way to do things while taking the API's limitations into account and adapting to the requirements and the new ideas that came up along the way was a difficult part of our work that will train us for an important part of our job.

As none of us had experience in writing WebExtensions, discovering the API and what could be done was also a large part of our work. Thunderbird's documentation isn't the most well-made and going through it was kind of a chore at times but having it as our only source of information required us to do something that was surprisingly rarely asked in classes despite how crucial using and understanding an official documentation is for developers. Even though Thunderbird's API differs on some aspects with standard WebExtensions, most of the knowledge we acquired with this project could probably be used if we ever wanted to build an extension for a web browser, which may be useful in the future as web services are taking an increasingly large amount of space in the software industry.

This project was also a good occasion to improve our skills in web languages, especially JS as most of the group wasn't familiar with it considering it was a very minor part of our web programming classes. Sharing our knowledge and discovering new aspects of Javascript programming increased everyone's level to a great extent.