What is email verification or validation and how does it work?

Posted on January 14, 2021

The verification or validation of email addresses is a fundamental need in many applications, ranging from protection against phishing and various other email-based threats through the validation of data filled in into an online form to the purification of marketing or other email lists from invalid addresses to maintain sender reputation and avoid bouncing or other unnecessary email traffic. These have been illustrated in detail in several other blogs.

In what follows we illustrate how an actual validation is carried out. But first of all, let's answer this question:

What is email verification?

By "validation" we mean a verification of whether the given email address exists and can receive emails, along with obtaining some other supplemental information about its nature. As such it is not to be confused with techniques to decide whether the address is malicious or blacklisted; that is a different deal.

The main purpose of this document is to give an overview of the relevant aspects of validation and the demonstration of how this can be done using the email verification API. This facilitates the validation of many email addresses quickly, one by one or in bulk, even in an automated fashion, possibly integrated into a bigger application.

But before we go for it, let's briefly recapitulate the basics of the email protocol. (If you are an expert on that, you may safely skip the next section.)

1. Background

The email protocol is among the oldest ones of the Internet still in operation. It was first specified as early as 1982 in RFC 821, and some of the details are still the same. The current specification, RFC 5321 also dates back to 2018 (albeit some of the details were updated in RFC 7504 in 2015. Altogether, like many other solutions of the Internet, email was designed with a relatively coherent and benevolent user community in mind. This is behind many of the issues necessitating email address validation and additional security measures.

In spite of the simplicity of the goal - send an electronic message to another user on the Internet -, the protocol is relatively complex, and its development turned the operation of an email server into an advanced matter of system administration.

From our point of view the relevant part is the way how an email is sent and received if no additional problem prevails. Assume that in a fictitious situation a sender, alice@example.com wants to send an email to the address bob@whoisxmlapi.com. Alice is supposed to have an account on the local system and she has an email client on her device (this is termed as a Mail User Agent or MUA) configured to use a mail server to send and receive messages. This server is called the Mail Transfer Agent (MTA). Upon pressing the "Send" button, her MUA will thus instruct the MTA to send the mail to the recipient to bob@whoisxmlapi.com. Here is what will happen in the simplest case:

- Alice's MTA shall look for the name and IP of the mail exchange server (MX) in the recipient domain. This can be queried in the Domain Name System (DNS). In our example, Bob's domain is whoisxmlapi.com, thus the MTA needs to query the DNS for whoisxmlapi.com's mail servers; these are the so-called MX (Mail Exchange) records in the DNS response. We can try this manually, e.g., from a current Linux or Mac OS X command-line by running

```
host -t mx whoisxmlapi.com
```

resulting in the following reply:

```
whoisxmlapi.com mail is handled by 30 alt2.aspmx.l.google.com.
whoisxmlapi.com mail is handled by 10 aspmx.l.google.com.
whoisxmlapi.com mail is handled by 20 alt1.aspmx.l.google.com.
whoisxmlapi.com mail is handled by 40 aspmx2.googlemail.com.
whoisxmlapi.com mail is handled by 50 aspmx3.googlemail.com.
```

The mail exchange servers' host names responsible for dealing with the mail traffic of that domain are nominated. To ensure availability and load balancing there are multiple servers. The numeric value between "by" and the hostname is termed as the "priority": the server with the lowest number should be preferred and if anything fails, the others should be tried to communicate with the next one, respectively. So Alice's MTA will use aspmx.l.google.com.

- Alice's MTA connects the recipient's MX via the Simple Mail Transfer Protocol (SMTP). This is a simple dialog between the two MTA's on the standard port of SMTP (normally port 25, sometimes 465 or 587). A typical conversation would look something like this in our case:

```
R: 220 aspmx.l.google.com ESMTP some_server_implementation
S: HELO mailserver.example.com
R: 250 mailserver.example.com, I am glad to meet you
S: MAIL FROM:<alice@example.com>
R: 250 Ok
S: RCPT TO:<bob@whoisxmlapi.com>
R: 250 Ok
S: DATA
R: 354 End data with <CR><LF>.<CR><LF>
S: From: "Alice Doe" <alice@example.com>
S: To: Bob Doe<bob@whoisxmlapi.com>
S: Date: Tue, 5 Jan 2021 16:02:43 -0500
S: Subject: Hello Bob
S:
S: Hello Bob,
S:
S: I'm sending this mail to remind you...
S: Yours
S:
S: Alice
```

```
S:
S: .
S:
R: 250 Ok: queued as 12345
S: QUIT
R: 221 Bye
```

Here R and S stand for "Receiver" and "Sender", and the example is somewhat artificial as actually Google's mail servers use different and less illustrative messages, however, the main elements here are the numeric codes sent from the receiver and the instructions with capital letters from the sender. What happened was that the sender said HELO (aka "Hello") to the server which then politely replied. After this handshake, the sender told that there is a mail message to be sent to the other side. The receiver said fine, and received the message. Then the connection terminated.

Such a communication can be carried out even manually, e.g., connecting to the respective port of the server, like this:

```
telnet aspmx.l.google.com 25
```

However, it will not always work out, and it is not always recommended to tamper with that. For instance, if you are sitting at a computer having some ordinary Internet access, your internet service provider is most probably blocking outbound communication on port 25 as normal users normally do not directly connect to MX servers and this possibility could be used for sending spam. (In fact this is the root cause of why we do not display an actual communication log above in our example.)

However, just as expected, it is not always so easy: this communication can fail in many ways; there is a tremendous number of the 3-digit return codes the receiver may respond with. For instance, our communication may have resulted in something like

```
550 5.1.1 <bob@whoisxmlapi.com>: Recipient address rejected: User unknown i
```

reflecting that the mail address is invalid.

- Finally the message ends up in Bob's mailbox, who will then view it with his favorite mail client (using the POP3 or the IMAP protocol, or some fancy webmail interface).

The above picture of the email protocol is very far from being comprehensive: we didn't mention relaying, repeated attempts to send an email, bouncing, etc. As this Section is not a book about email technology, just a humble summary of the basic ideas necessary to understand how email verification works, it is time now to turn our attention to this original topic.

2. Aspects of email verification

Let us now go quickly through the logical steps of email address validation; these are all implemented in the Email Verification API.

2.1 Syntax check

The syntax validation of emails is a frequently mentioned textbook example of using regular expressions. There are simpler approaches that may fail on some otherwise valid email addresses, and there are ones which perform very well. For sure the syntax validation can be done locally with any tool that supports regular expressions. Nevertheless the email verification API has an elaborate regular expression built in, and it will let us know if the string we validate can be an email at least syntax-wise.

If we send an email address to the API that does not meet the syntax requirements, it will return an error message "The email address must be a valid email address". (Note that there is a legacy formatCheck field for backward compatibility reasons in the API output, but it always has a true value.)

2.2. DNS check

Recall that the first step by the MTA is to find out the name of the appropriate MX server. Thus the

first step should be to query the MX server of the domain name in the email, "whoisxmlapi.com", which can even be done manually as described in the previous section.

It can be the case though that the domain does not at all exist, so certainly the domain itself should be valid. This can be easily found out with a DNS lookup: a nonexistent domain will result in an NXDOMAIN reply of the host command. If once we are to validate an email address, however, the API will of course do this for us, and there will be two types of information of this kind in the results:

- dnsCheck will be true if the domain exists, and false otherwise.
- mxRecords will provide the actual list of mail servers. If the DNS check returns "true" while this list is empty or has a single element, ".", it means that even though the domain itself exists, it is not configured for email exchange.

2.3 SMTP check

If the domain exists and there are MX servers nominated, the next step is to check if any of them can actually be connected via SMTP. This is done by initiating a communication and eventually imitating a mail sending operation to the address being validated.

Again, this can be done manually, e.g., by connecting the servers via telnet as described above and doing the appropriate conversation. However, as said before, it might be impossible from a workstation's IP because of firewall rules of our network. Also, there is a risk that our simulated email sending communication is found suspicious by some protective automatism on the mail server's side, especially when we do it for many addresses in bulk. As a negative consequence our IP can get blacklisted.

Luckily, the API does this favor for us. In the reply, the field smtpCheck will be true if at least one MX server could be successfully connected and it was ready to receive emails for the address. The field's value will be false otherwise.

2.4 Catch-all address check

At this point we can be almost sure that an email sent to the address being validated will be

actually taken by the MX server. On the receiver side, it should end up in a mailbox. While it is not possible to find out whether the mail will end up in a spam folder (this is done entirely in the receiver's system internally), we can still get some information about the nature of the mailbox.

Namely, if we send an address to an invalid email account in a domain, normally our message bounces back with an error that should have been already detected by the SMTP check. There is another option, too. Some MX servers accept all inbound messages but those which belong to a nonexistent account will end up in one or more accounts termed as the "catch-all email addresses".

To detect this, one just has to verify the existence of one or more email addresses with a randomly generated account name in the given domain: if they work, there should be a catch-all address there. The API will do this for us, too: the output field catchAllCheck will be "true" if and only if the email address' domain has a catch-all address.

2.5 Mail provider type check

It may also be of interest whether the email address belongs to a free email service provider like Yahoo! or Google Mail, or it is run by some entity (like a company, academic organization, etc.) who does not give an email address to anyone upon request. The way to determine that is to look the address up in various databases, and indeed, the API will do this, too. The result field freeCheck will let us know.

2.6 Disposable email check

Another aspect is the question of disposable email addresses. There are email providers who offer short-lived email addresses. These are used, for instance, by those who want to use services which require registration, but do not expect any email correspondence from there, nor do they want to receive any marketing material. There are databases about such providers, one of themost comprehensive is also available for purchase as a whole. The Email Verification API will lookup the mail address to be checked in this database and return a true~/~false value in the disposableCheck field.

3. Using the API

Let us now see a few examples of how the API works. Being a standard RESTful API capable of producing XML and JSON output, it can be easily invoked with the curl utility in its simplest form:

```
curl --get  "https://emailverification.whoisxmlapi.com/api/v1?apiKey=YOUR_A
```

Here "YOUR_API_KEY" should be replaced with your API key, just subscribe for a free access. We have added | jq to get a nicely tabulated and colored terminal output, it can be omitted if jq is not installed on your system or you prefer raw JSON output. In this example we try validating not.a.valid_email as an email address which is obviously invalid. Indeed, we get:

```
{
 "ErrorMessage": [
   {
     "Error": "The email address must be a valid email address."
   }
 ]
}
```

Let us now try checking a syntactically valid and existing mail address:

```
curl --get  "https://emailverification.whoisxmlapi.com/api/v1?apiKey=YOUR_A
```

resulting in

```
{
 "emailAddress": "support@whoisxmlapi.com",
 "formatCheck": "true",
 "smtpCheck": "true",
 "dnsCheck": "true",
 "freeCheck": "false",
 "disposableCheck": "false",
 "catchAllCheck": "true",
 "mxRecords": [
   "alt1.aspmx.l.google.com.",
   "aspmx2.googlemail.com.",
   "aspmx.l.google.com.",
   "aspmx3.googlemail.com.",
   "alt2.aspmx.l.google.com."
 ],
 "audit": {
   "auditCreatedDate": "2021-01-06 15:01:54.000 UTC",
   "auditUpdatedDate": "2021-01-06 15:01:54.000 UTC"
 }
}
```

As you can see, "support@whoisxmlapi.com" is a well-formatted and properly configured, existing email address, which is not at a free provider and is not disposable. The audit dates reflect when the validation actually took place. Indeed, this is our support email address waiting for your requests. Also, we do have a catch all address, we shall see a consequence of this later.

Checking one of our earlier example addresses, bob@whoisxmlapi.com, we shall have

```
{
 "emailAddress": "bob@whoisxmlapi.com",
 "formatCheck": "true",
```

```
  "smtpCheck": "true",
  "dnsCheck": "true",
  "freeCheck": "false",
  "disposableCheck": "false",
  "catchAllCheck": "true",
  "mxRecords": [
    "alt1.aspmx.l.google.com.",
    "aspmx2.googlemail.com.",
    "aspmx.l.google.com.",
    "aspmx3.googlemail.com.",
    "alt2.aspmx.l.google.com."
  ],
  "audit": {
    "auditCreatedDate": "2021-01-06 16:02:03.953 UTC",
    "auditUpdatedDate": "2021-01-06 16:02:03.953 UTC"
  }
}
```


Don't be surprised: had we got a colleague named "Bob" with this address, we wouldn't have used his actual address here as an example. We do have, however, a catch-all address: we do not drop any inbound mails thus any syntactically valid "username" can receive emails. We collect these mails in a dedicated mailbox. Consequently in the domain whoisxmlapi.com, which is properly configured in the DNS otherwise, the smtpCheck will always be true.

In contrast to that, for instance, the Massachusetts Institute of Technology does not have a catch-all address which is rather common in case of academic establishments. Hence, if we check, e.g., dsaascsacsad@mit.edu which is not likely to belong to an existing user, we get


```
{
  "emailAddress": "dsaascsacsad@mit.edu",
  "formatCheck": "true",
  "smtpCheck": "false",
  "dnsCheck": "true",
  "freeCheck": "false",
  "disposableCheck": "false",
  "catchAllCheck": "false",
```

```
  "mxRecords": [
    "mit-edu.mail.protection.outlook.com."
  ],
  "audit": {
    "auditCreatedDate": "2021-01-06 19:04:55.213 UTC",
    "auditUpdatedDate": "2021-01-06 19:04:55.213 UTC"
  }
}
```


Observe that smtpCheck is false in this case: In contrast to that, for admissions@mit.edu, one of their published email contacts from their web page we get


```
{
  "emailAddress": "admissions@mit.edu",
  "formatCheck": "true",
  "smtpCheck": "true",
  "dnsCheck": "true",
  "freeCheck": "false",
  "disposableCheck": "false",
  "catchAllCheck": "false",
  "mxRecords": [
    "mit-edu.mail.protection.outlook.com."
  ],
  "audit": {
    "auditCreatedDate": "2021-01-06 19:08:31.951 UTC",
    "auditUpdatedDate": "2021-01-06 19:08:31.951 UTC"
  }
}
```


While we will not demonstrate the detection of a free email provider because of privacy reasons, let us now demonstrate how a disposable email is detected. We have generated a disposable email address at and its check has resulted in the following:


```
{
  "emailAddress": "carling@spybox.de",
```

```
  "formatCheck": "true",
  "smtpCheck": "true",
  "dnsCheck": "true",
  "freeCheck": "false",
  "disposableCheck": "true",
  "catchAllCheck": "false",
  "mxRecords": [
    "tempmail.de."
  ],
  "audit": {
    "auditCreatedDate": "2021-01-06 22:34:10.000 UTC",
    "auditUpdatedDate": "2021-01-06 22:34:10.000 UTC"
  }
}
```


There can be of course many examples, for further details we refer to the API docs. A bunch of integrations: code examples for popular development environments is also available. It is also worth mentioning that in the need of the validation of a larger number of email addresses at once, there is a separate bulk email lookup API which has the same business logic for individual validation but performs efficiently on a list of many emails.