What Is Email Verification or Validation and How Does It Work?

Posted on January 14, 2021

Email address verification or validation is fundamental to many business processes, especially marketing. Validation begins when users fill in online forms and continues on until filtering invalid email addresses out of marketing or other lists to maintain one's good sender reputation, avoid bounces, and block malicious messages, as illustrated in greater detail in several other posts.

We'll illustrate how email validation is actually carried out in the next sections, but let's start with the basics first.

What Is Email Verification?

By "verification," we mean validating if a given email address exists and can receive messages. The process also involves obtaining other supplemental information about an email address's nature. It should not be confused with techniques to decide if the email address is malicious or blacklisted, as that is an entirely different thing.

This post aims to present an overview of the relevant aspects of validation and a demonstration of how that is done using an email verification API. The tool facilitates the verification of several email addresses quickly, whether one by one or in bulk in an automated fashion, and possibly even within a bigger application. But before we go into the nitty-gritty of email validation, let's review the email protocol basics first. (If you are an expert on that, you may safely skip the next section.)

1. Background

The email protocol is one of the first Internet standards that remain in use until now. While it was

developed as early as 1982 and specified in RFC 821, many of its details haven't changed since. Over time, the protocol has been updated twice—once in 2015 (RFC 7504) and again in 2018 (RFC 5321). Like many other Internet technologies, emailing can be riddled with challenges, necessitating email address validation and additional security measures.

Despite the simplicity of email's goal—to send a message to another user electronically via the Internet—the protocol is relatively complex, turning the operation of an email server into an advanced matter of system administration.

For general users, what matters is how an email gets sent and received, regardless of issues encountered. Let's take a look at an example. Imagine that a user (alice@example[.]com) wants to send an email to the address bob@whoisxmlapi[.]com. Alice needs an email account and an email client on her device, collectively known as a "Mail User Agent (MUA)," configured to use a mail server to send and receive messages. The server in this case is called the "Mail Transfer Agent (MTA)." Once Alice hits the "Send" button, her MUA will instruct the MTA to send her mail to the recipient (bob@whoisxmlapi[.]com). The following process explains what happens in the background in greater detail:

- Alice's MTA will look for the name and IP address of the mail exchanger (MX) using the recipient's domain name as reference. The MTA will need to query the Domain Name System (DNS). In our example, Bob's domain is whoisxmlapi[.]com so the MTA needs to query the DNS for whoisxmlapi[.]com's mail servers. The MX server looks at MX records to give the correct DNS response. You can try doing that manually on a Linux or macOS computer using the following command:

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

You will get the following response:

```
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 response gives the hostnames of the MX servers responsible for dealing with the domain's mail traffic. To ensure availability and load balancing, each domain has multiple servers. The numeric value between the word "by" and the hostname is known as the "priority." The server with the lowest number gets contacted first and if it fails to respond, the others are accessed sequentially.

In the example above, Alice's MTA will use "aspmx.1.google.com" first then "alt1.aspmx.l.google.com" and so on.

- Alice's MTA connects with the recipient's MX server via the Simple Mail Transfer Protocol (SMTP). They exchange a simple dialog via the standard port assigned to SMTP traffic (normally port 25, sometimes 465 or 587). A typical conversation would look like this:

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

In the sample dialog above, “R” and “S” stand for “Receiver” and “Sender,” respectively. The example is somewhat artificial, as Google’s mail servers use different and less illustrative messages. The main elements are the numeric codes sent to the receiver and the instructions in capital letters from the sender. What happened was the sender said HELO (aka “Hello”) to the server, which then politely replied. After this handshake, the sender told the server it has a mail message to send to the receiver. The receiver said fine and got the message before the connection got terminated.

Such a communication can be carried out manually by connecting to the server port using a command like:

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

This action will not always work nor is it recommended. If not done right, it might tamper with how the computer works. The problem may also rest on the part of the Internet service provider (ISP). It could be blocking outbound communication on port 25 since normal users don’t directly connect to MX servers, as they could get mistaken for spammers.

There are other reasons why this kind of communication fails. That is why there is a tremendous number of three-digit return codes the receiver may respond with. One example would be:

550 5.1.1 <bob@whoisxmlapi.com>: Recipient address rejected: User unknown in virtual mailbox tabl

This response means the mail address is invalid.

- As the final step, the message ends up in Bob's mailbox. He can view the email using his preferred mail client via POP3 or IMAP and a fancy webmail interface.

The email protocol, as described here, is far from comprehensive. We didn't mention how relaying works and what bouncing means, among others. The contents of this section comprises a tiny portion of a book about email technology. It only provides a humble summary of the basic ideas you'll need to understand how email verification works but we'll try to explain the process as simply as possible.

2. How Does Email Verification Work?

Email verification is the process of validating messages by following a series of steps that help users improve their email deliverability. The general process has five steps but the more comprehensive the email verification tool is, the more steps and checks there are.

First, an email validation tool identifies issues and problems before ISPs or email service providers (ESPs) catch them. It checks an email address's spelling. Companies that buy mailing lists always have a few invalid and unreachable email addresses in their mailing lists. Sending messages to these can be a cause for ISP or ESP blacklisting. The tool also checks an email address's format, otherwise users can suffer from hard bounces. Then it verifies if the domain has DNS records and that its MX server can receive emails. Finally, the tool checks for mailbox presence using SMTP.

Let's now dive into each aspect of email verification in greater detail now.

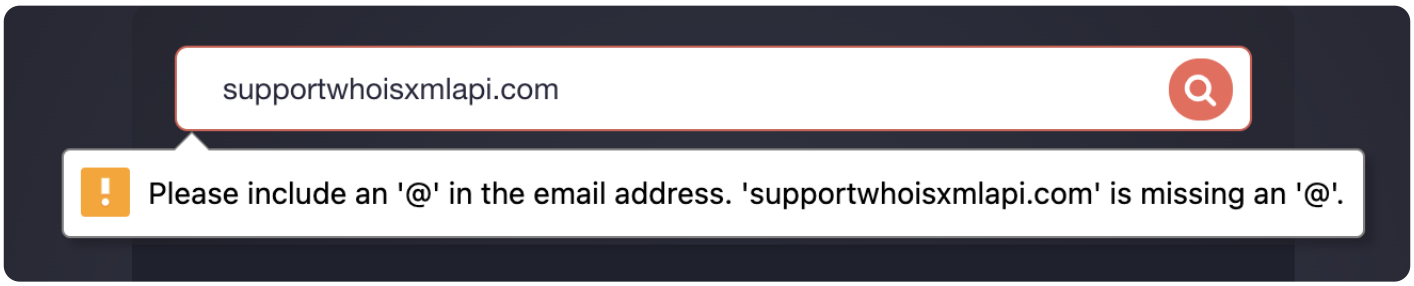
3. Aspects of email verification

Note that each step of email address validation is done by Email Verification API.

3.1 Syntax check

Syntax validation, as frequently mentioned in textbooks, uses regular expressions, which may either fail or work very well. The process can be done locally using any tool that supports regular expressions but our email verification API has a built-in elaborate regular expression that will let users know if the string being checked can be a proper email address at least based on syntax.

Querying an email address that doesn't meet the required syntax on the API will cause it to return the following result:

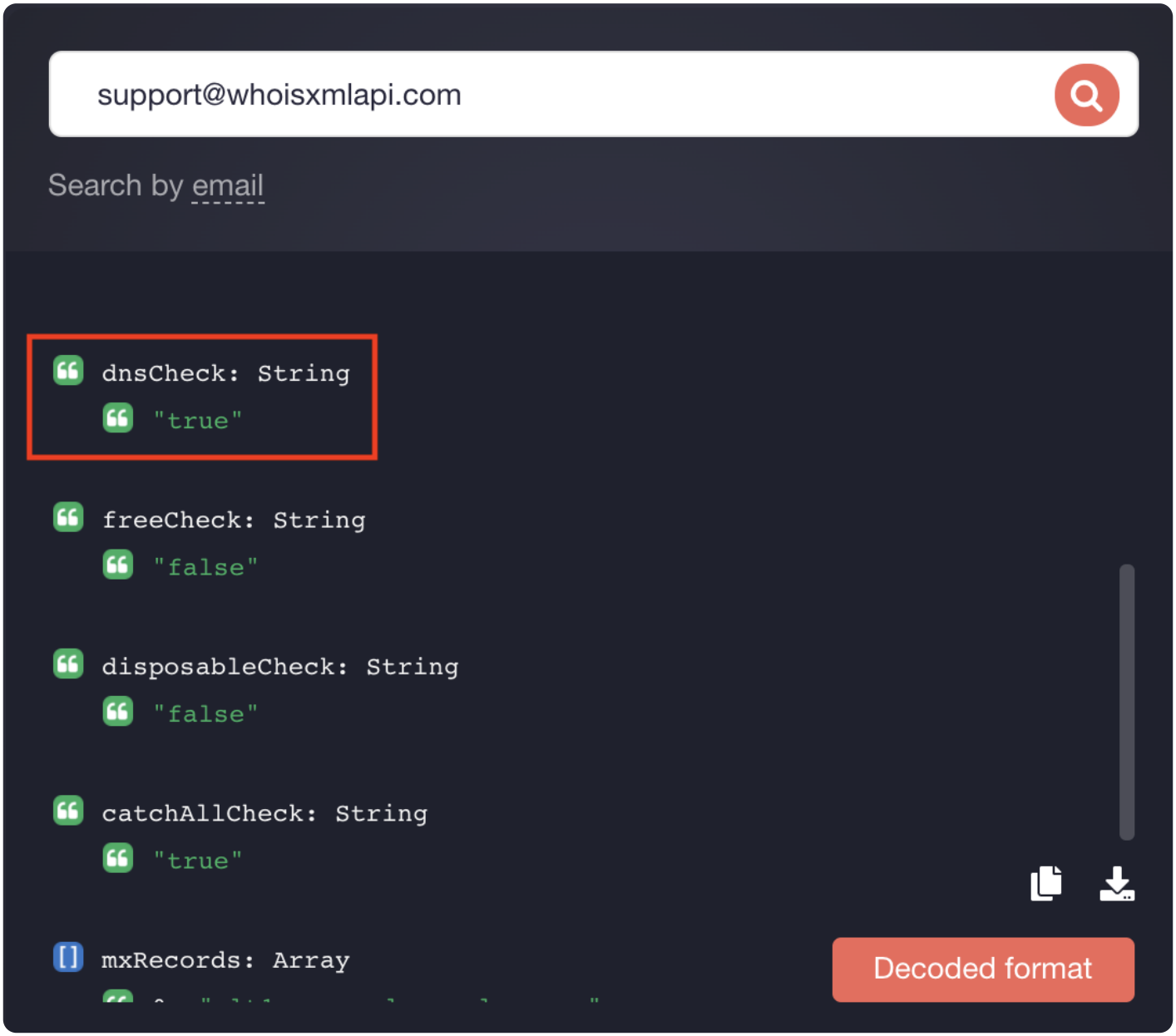


3.2. DNS check

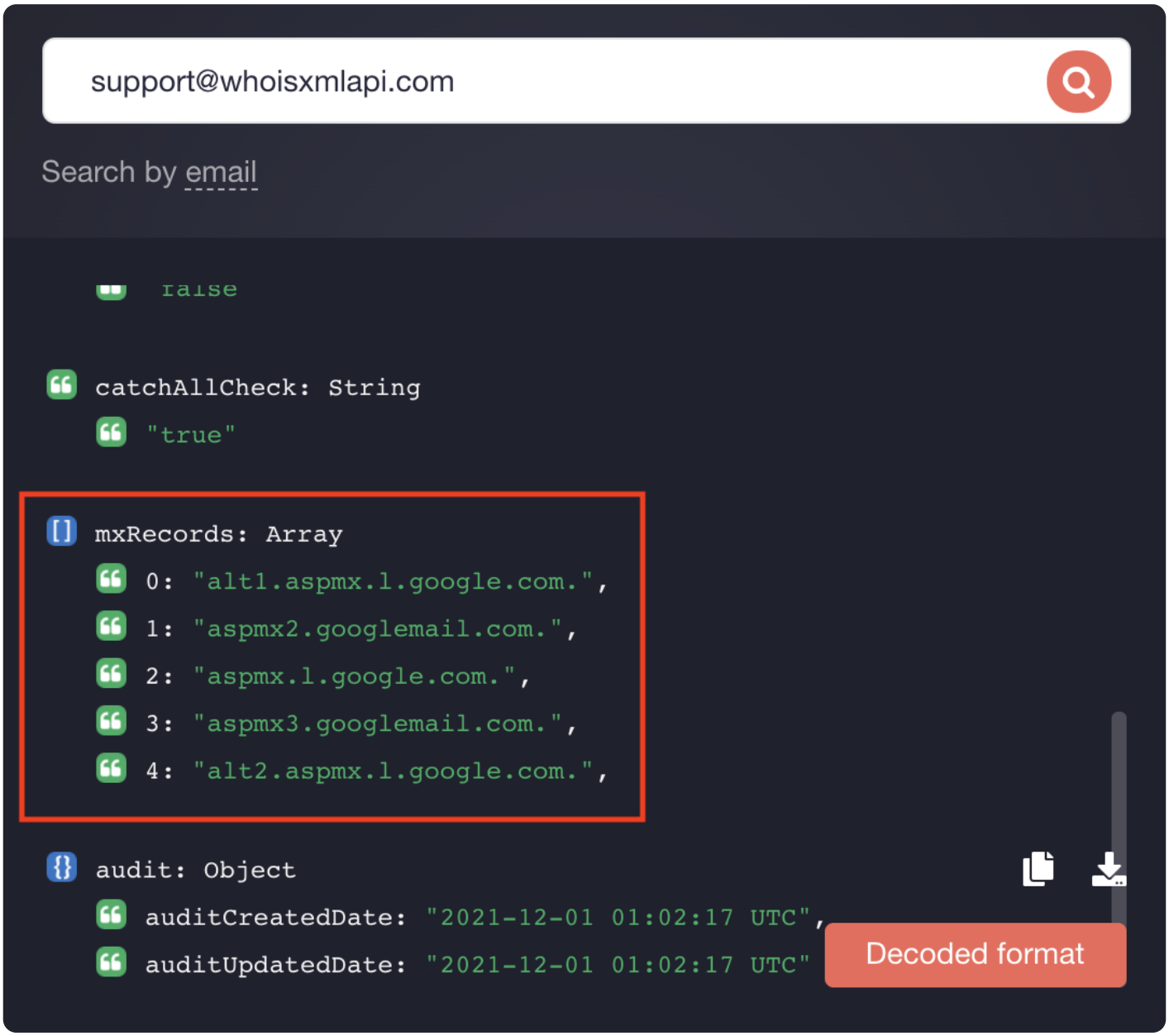
Recall that the first step an MTA takes is to identify the name of the email address's appropriate MX server. As such, the API queries the MX server of the domain name—whoisxmlapi.com. It is possible that the domain does not exist so it would be tagged "invalid."

Using the API, of course, does the check more quickly and provides more information:

- If the domain exists, the dnsCheck will reply with "true," if it doesn't, you'll get "false" as a response.



- The API will also provide a list of the domain's MX servers under "mxRecords: Array." If the DNS check result is "true" but the list is empty or only has one string ("."), the domain exists but is not configured for email exchange. A domain that is properly configured should show this response:

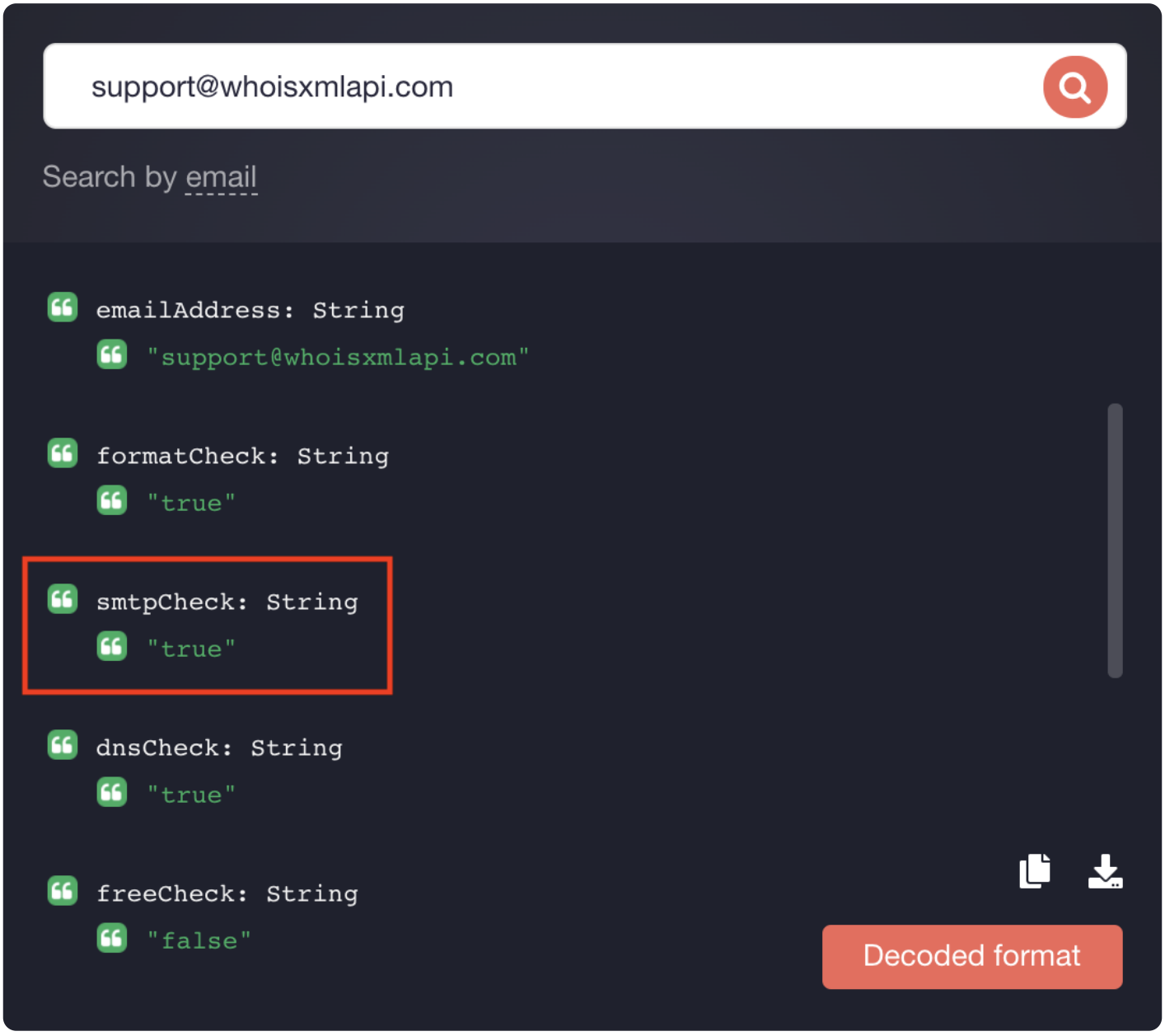


3.3 SMTP check

If the domain exists and has MX servers, the API checks if any of them can actually be accessed via SMTP. That is done by initiating communication and imitating a mail-sending operation to the address undergoing validation.

An SMTP check can be done manually by connecting to the server via telnet as described earlier and going through the appropriate conversation. But as we said, it may be impossible if the network is firewall-protected. There is also a risk for the simulated email-sending communication to be found suspicious by protective automatisms on the mail server's side, especially if the process is performed for addresses in bulk. If that happens, the user’s IP address can get blacklisted.

Luckily, the API does the SMTP check in a way that the user’s domain can’t get blacklisted. It will give the following result if the connection is successful, signifying that the email address can receive messages:

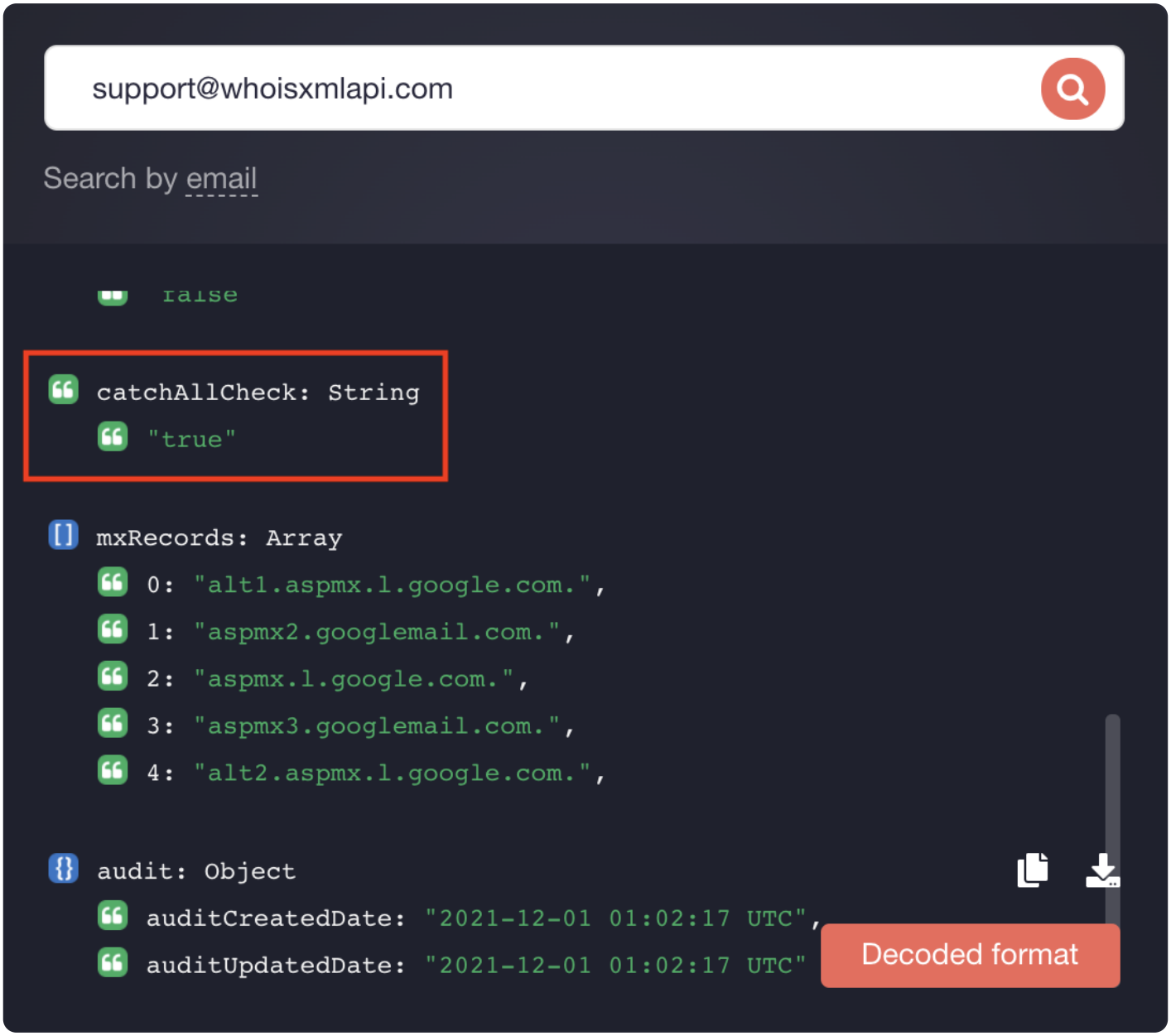


3.4 Catch-all address check

At this point, we can almost be sure that an email sent to the address undergoing validation will be accepted by the MX server. On the receiver's side, it should end up in his/her mailbox. While itisn't possible to know if the mail will end up in the spam folder or not, we can still get someinformation about the nature of the mailbox.

If users send messages to an invalid email domain, they will bounce. That could have been prevented by the SMTP check. There is, however, another option. Some MX servers accept all inbound messages but those meant for nonexistent accounts end up in a so-called "catch-all email account."

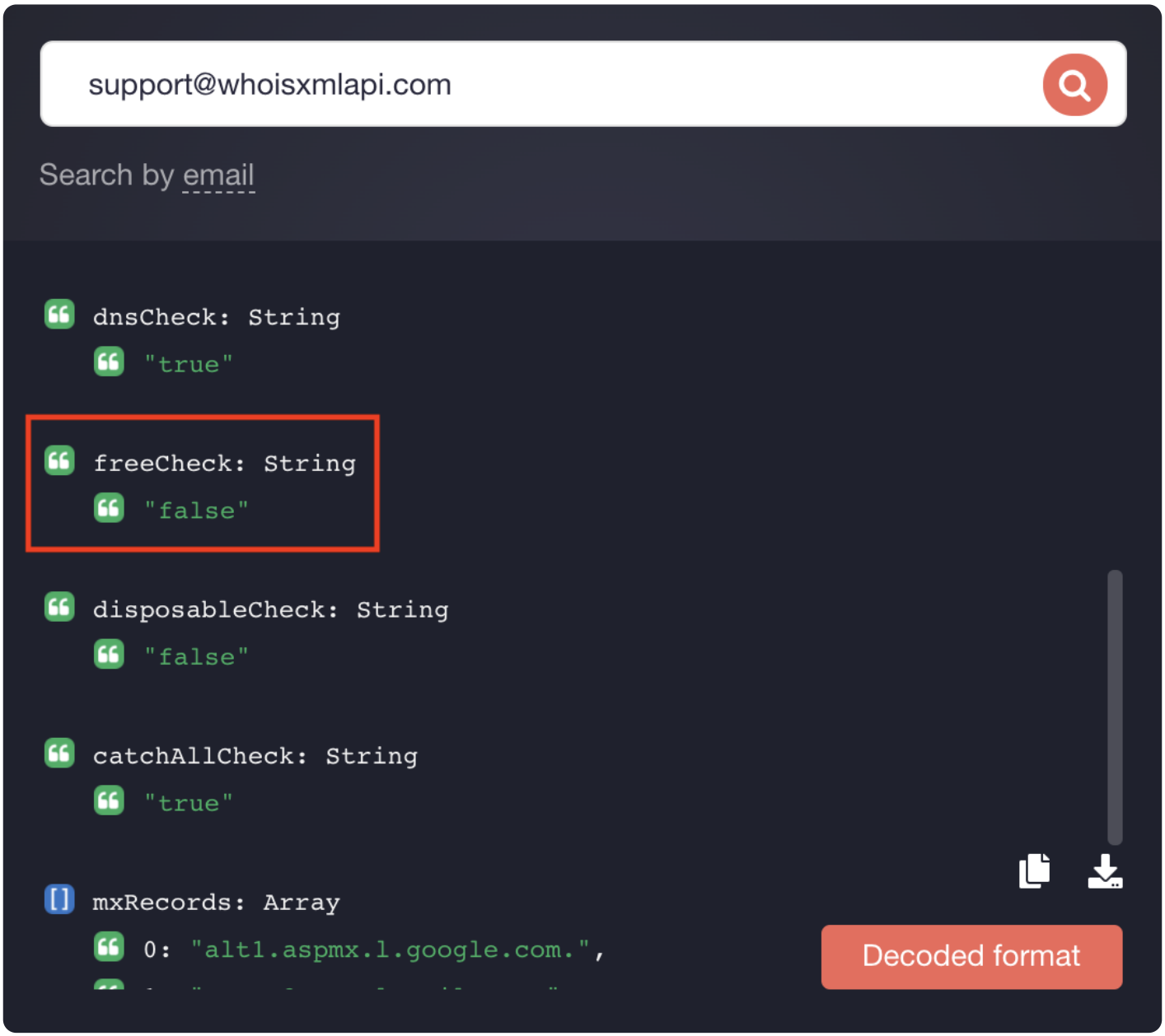
Users can detect the existence of a catch-all email address via an email verification API catch-all check. A domain with a catch-all email address gives this result:



3.5 Mail provider type check

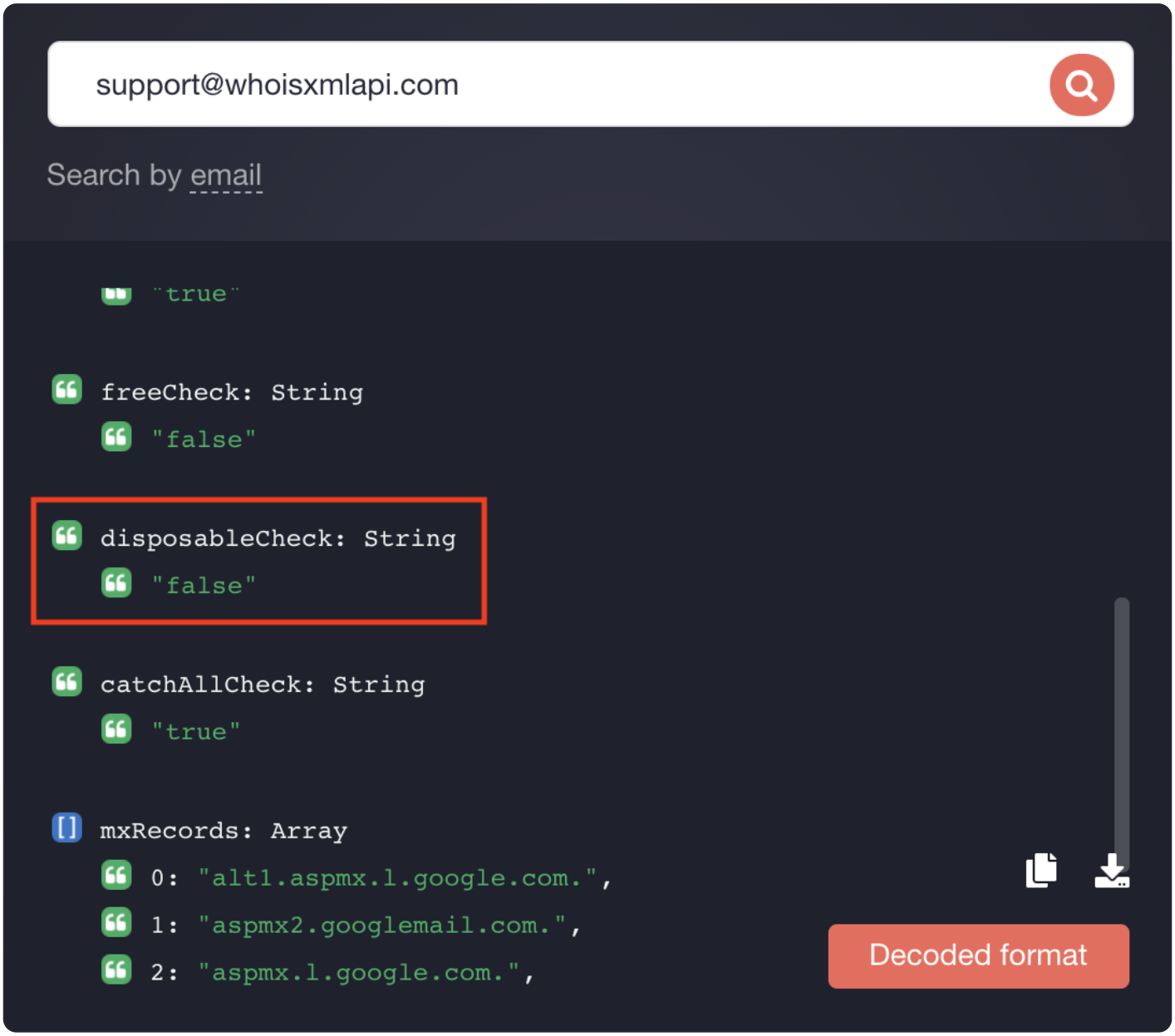
Users may also find it interesting to know if an email address belongs to a free email service provider like Yahoo! or Gmail or a particular company, school, or any other organization. One way

to do that is to look the email address up in various databases but a far faster manner would be to let the API do that. An email address that belongs to a specific organization and not a free service gives this result:



3.6 Disposable email check

Another important aspect to consider is if the email address is disposable. Many email service providers offer temporary email addresses, which are generally used by those who don't want to fill up registration forms. They don't expect to receive any email correspondence from the disposable email address. They only created the address for marketing materials. While users can buy disposable email databases from many providers, ours is one of the most comprehensive. Those who don't want to go through such a database manually can, however, rely on Email Verification API. A primary email address or one that the intended recipient most likely uses actively should give this result:



4. Using the API

Let's now look at a few examples to demonstrate how the API works. As a standard RESTful API capable of producing XML and JSON outputs, users can easily invoke the tool using the curl utility in its simplest form:

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

Replace “YOUR_API_KEY” with your API key by subscribing for free. We added “| jq” to give you a nicely tabulated and colored terminal output. You can omit it if jq is not installed on your system or you prefer to see a raw JSON output. In this example, we tried validating “not.a.valid_email” as an email address, which is obviously invalid. So we got this result:


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


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

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

That resulted 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 properly formatted and configured existing email address. It is not hosted by a free provider and is not disposable. The audit dates reflect when the validation took place. The email address is actually our support email address, which accepts requests from all our subscribers. We have a catch-all email address, too.

Checking the sample email address bob@whoisxmlapi.com, we got the result:


```
{
  "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"
 }
}
```


Users shouldn't be surprised because we do have an employee named "Bob" but we didn't use his full email address in this post (in other words, that isn't his correct email address). But since we have a catch-all address, your message to Bob would end up in our catch-all mailbox.

The domain whoisxmlapi.com is also properly configured and has DNS records, as such the email address bob@whoisxmlapi.com will pass the SMTp check.

In contrast, the Massachusetts Institute of Technology (MIT) does not have a catch-all address, which is rather common for academic institutions. As such, if we check dsaascsacsad@mit.edu, which most likely doesn't belong to an existing user, we will get this result:


```
{
 "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"
 }
```

```
}
```


Notice that the SMTP check gave “false” as a response. Checking admissions@mit.edu, one of MIT’s published email addresses on its web page, meanwhile, gave this result:


```
{
 "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 won’t demonstrate how to check for free email providers due to privacy reasons, we’ll show how disposable addresses are detected. We generated a disposable email address and checked it. We got this result:


```
{
 "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 are many other examples. You can check out the API docs for more information. A bunch of integrations, including code examples for popular development environments, is also available. It is also worth mentioning that users who need to validate a large number of email addresses at once can use Bulk Email Verification Lookup.