How to parse a json file from Linux command line using jq

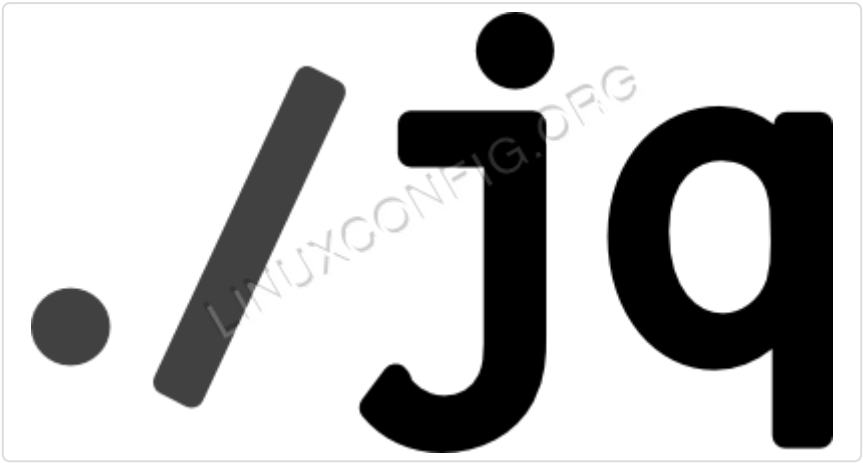
Egidio Docile Programming & Scripting 02 June 2020

The `JSON` (JavaScript Object Notation) format is widely used to represent data structures, and is frequently used to exchange data between different layers of an application, or by the use of API calls. We probably know how to interact with json-formatted data with the most used programming languages such as parsing JSON with python, but what if we need to interact with it from the command line, or in a bash script? In this article we will see how we can accomplish such a task by using the `jq` utility and we will learn its basic usage.

Contents

In this tutorial you will learn:

- How to install jq in the most used Linux distributions or compile it from source
- How to use jq to parse json-formatted data
- How to combine filters using "," and "|"
- How to use the length, keys, has and map functions



Software Requirements and Conventions Used

Software Requirements and Linux Command Line Conventions

Category	Requirements, Conventions or Software Version Used
System	Distribution-independent
Software	The jq application
Other	Familiarity with JSON data and the bash shell

Category	Requirements, Conventions or Software Version Used
Conventions	# - requires given linux commands to be executed with root privileges either directly as a root user or by use of `sudo` command **$** - requires given linux commands to be executed as a regular non-privileged user

Installation

The `jq` utility is included in all the major Linux distributions repositories, therefore installing it is very easy: we just need to use our favorite package manager. If we are using Debian, or a Debian-based distribution such as Ubuntu or Linux Mint, we can use `apt`:

```
$ sudo apt install jq
```


If we have a preference for the Red Hat family of distributions, such as Fedora, CentOS or RHEL, we can install `jq` via the `dnf` package manager (in recent versions of those distributions it superseded yum). To install the package we would run:

```
$ sudo dnf install jq
```

Installing `jq` on Archlinux is just as easy. The distribution package manager is `pacman`, and the package is available in the community repository. We can perform the installation with the following command:

```
$ sudo pacman -S install jq
```

If we can't, or for some reason we don't want to use a pre-built binary package, we can compile jq from source. In the following lines we describe the needed steps.

Building and installing from source

To build and install jq from source, the first thing we must do is to download a release tarball. At the moment of writing, the latest available release is `1.6`. To download the tarball without leaving the terminal, we can use `wget`:

```
$ wget https://github.com/stedolan/jq/releases/download/jq-1.6/jq-1.6.tar.gz
```

Once the download is complete, we must decompress and extract the tarball:

```
$ tar -xzf jq-1.6.tar.gz
```

The next step is to enter the `jq-1.6` directory, created as a result of the last command:

```
$ cd jq-1.6
```

Now, to compile the source code we need the following utilities:

- gcc
- automake
- libtool
- make

To build the software we run:

```
$ autoreconf -fi
$ ./configure && make && sudo make install
```

The `make install` command, by default, will cause binaries to be installed in the `/usr/local/bin` directory, and libraries into `/usr/local/lib`. If we want to customize the installation, and change those directories, we must specify a different prefix, using the `--prefix` option when launching the `./configure` script.

For example, to install the software only for a specific user, we could pass the `$HOME/.local` directory as prefix: in that case binaries would be installed into `$HOME/.local/bin` and libraries into the `$HOME/.local/lib`; with such configuration there would be no need to launch the `make install` command with administrative privileges. If you want to know how to better organize software installed form source, you can check our article about the GNU stow utility.

Usage

Once we have `jq` installed, we can use it to parse json files from the command line. For the sake of this tutorial we will work with a simple data structure which contains some details about three characters from Lord Of The Rings book. The data is saved in to the `characters.json` file.

The `jq` utility works by applying filters on a stream of json data. As a first thing, we will use the most simple filter, `.`, which returns the input data unchanged but pretty printed. For this characteristic, it can be used to format data in a more readable way:

```
$ jq . characters.json
```

The command above produces the following output:

```
{
  "characters": [
    {
      "name": "Aragorn",
      "race": "man"
    },
    {
      "name": "Gimli",
      "race": "dwarf"
    },
    {
      "name": "Legolas",
      "race": "elf"
    }
  ]
}
```

Now, suppose we want to filter the data to obtain only the value associated to the `characters` key. To accomplish the task, we provide the name of the key, and obtain its value (or `null` if it doesn't exist):

```
$ jq .characters characters.json
```

In our example the value associated with the "characters" key is an `array`, so we obtain the following result:

```
[
  {
    "name": "Aragorn",
    "race": "man"
  },
  {
    "name": "Gimli",
    "race": "dwarf"
  },
  {
    "name": "Legolas",
    "race": "elf"
  }
]
```

What if we want to get only the first element of the array? We just need to "extract" the right index from it. Knowing that arrays are `zero-based`, we can run:

```
$ jq .characters[0] characters.json
```

The command gives us:

```
{
  "name": "Aragorn",
  "race": "man"
}
```

We can also obtain a slice of the array. Say, for example, we want to get only its first two elements. We run:

```
$ jq .characters[0:2] characters.json
```

The command gives us the following result:

```
[
  {
    "name": "Aragorn",
    "race": "man"
  },
  {
    "name": "Gimli",
    "race": "dwarf"
  }
]
```

Slicing works also on strings, so if we run:

```
$ jq .characters[0].name[0:2] characters.json
```

We obtain a slice (the first two letters) of the "Aragorn" string: `"Ar"`.

Access array elements separately

In the examples above we printed the content of the "characters" array, which consist of three objects that describe fantasy characters. What if we want to iterate over said array? We must make so that elements contained in it are returned separately, so we must use `[]` without providing any index:

```
$ jq .characters[] characters.json
```

The output of the command is:

```
{
  "name": "Aragorn",
  "race": "man"
}
{
  "name": "Gimli",
  "race": "dwarf",
  "weapon": "axe"
}
{
  "name": "Legolas",
  "race": "elf"
}
```

In this case we obtained 3 results: the objects contained in the array. The same technique can be used to iterate over the values of an object, in this case the first one contained in the "characters" array:

```
$ jq .characters[0][] characters.json
```

Here we obtain the following result:

```
"Aragorn"
"man"
```

The "," and "|" operators

The "," and "|" operators are both used to combine two or more filters, but they work in different ways. When two filters are separated by a comma they are both applied, separately, on the given data and let us obtain two different results. Let's see an example:

```
$ jq '.characters[0], .characters[2]' characters.json
```

The json-formatted data contained in the characters.json file is first filtered with `.characters[0]` and then with `.charaters[2]`, to get the first and the third element of the "characters" array. By executing the command above, we obtain two **separate** results:

```
{
  "name": "Aragorn",
  "race": "man"
}
{
  "name": "Legolas",
  "race": "elf"
}
```

The "|" operator works differently, in a fashion similar to an unix pipe. The output produced by the filter to the left of the operator, is passed as input to the filter at the right of the operator. If a filter to the left of the operator produces multiple results, the filter to the right of the operator is applied to each one of them:

```
$ jq '.characters[] | .name' characters.json
```

In this example we have two filters. At the left of the operator we have the `.characters[]` filter, which as we previously saw, let us obtain the elements of the "characters" array as separate results. In our case, each result is an object with the `"name"` and `"race"` properties. The `.name` filter at the right of the `|` operator is applied to each one of the objects, so we obtain the following result:

```
"Aragorn"
"Gimli"
"Legolas"
```

Functions

The jq utility includes some very useful functions we can apply to the json - formatted data. We will now see some of them: `length`, `keys`, `has` and `map`.

The length function

The first one we will talk about is `length`, which, as the name suggests, let us retrieve the length of objects, arrays and strings. The length of objects is the number of their key-value pairs; the length of arrays is represented by the number of elements they contain; the length of a string is the number of characters it is composed of. Let's see how to use the function. Suppose we want to know the length of the "characters" array, we run:

```
$ jq '.characters | length' characters.json
```

As expected, we obtain `3` as result, since it is the number of elements in the array. In the same way, to obtain the length of the first object in the array we could run:

```
$ jq '.characters[0] | length' characters.json
```

This time we obtain `2` as result, since it is the number of value pairs contained in the object. As we alredy said, the same function applied to a string, returns the number of characters contained in it, so, for example, running:

```
$ jq '.characters[0].name | length' characters.json
```

We receive `7` as result, which is the length of the "Aragorn" string.

The keys function

The `keys` function can be applied on objects or arrays. In the first case it returns an array containing the objects keys:

```
$ jq '.characters[0] | keys' characters.json
[
  "name",
  "race"
]
```

When applied to an array, it returns another array containing the indices of the first one:

```
$ jq '.characters | keys' characters.json
[
  0,
  1,
  2
]
```

The `keys` function returns the elements sorted: if we want the elements to be returned in insertion order, we can use the `keys_unsorted` function instead.

Checking if an object has a key

One very common operation we may want to perform on an object, is checking if it contains a specific key. To accomplish this task we can use the `has` function. For example, to check if the main object of our json-formatted data contains the "weapons" key, we could run:

```
$ jq 'has("weapons")' characters.json
false
```

In this case, as expected, the function returned `false` since the object contains only the "characters" key:

```
$ jq 'has("characters")' characters.json
true
```

When applied to arrays, the function returns true if the array has an element at the given index or false otherwise:

```
$ jq '.characters | has(3)' characters.json
false
```

The "characters" array has only 3 elements; arrays are zero-indexed, so checking if the array as an element associated with the index `3` returns `false`.

The map function

The map function let us apply a filter to each element of a given array. For example, say we want to check the existence of the "name" key in each of the objects contained in the "characters" array. We can combine the `map` and `has` functions this way:

```
$ jq '.characters | map(has("name"))' characters.json
[
  true,
  true,
  true
]
```

Conclusions

In this article we barely scratch the surface of the features offered by the `jq` utility which let us parse and manipulate json-formatted data from the command line. We learned the basic usage of the program, how the "," and "|" operators work, and how to use the length, keys, has, and map functions, to respectively obtain the lenghts of arrays, strings and objects, obtain object keys or array indexes, check if a key exists in an object or if an array has an element at the given index, and apply a filter or a function to each element of an array. To discover all `jq` can do, go and take a look at the program manual!