**JQ Command in Linux with Examples**

JSON is a data representation format that is used to store and transfer data between different layers of an application; it stores data in key: value pairs.

The syntax of JSON was derived from JavaScript but it itself is language independent. It is compatible with many programming languages; these languages include code that can be used to integrate JSON into the program; but unfortunately, we cannot work with JSON directly in Linux shell as it cannot interpret it. To work with JSON in the Linux shell we use a mixture of tools such as JQ and sed.

In this post, we will learn to use the JQ command to manipulate and work with JSON data in a Linux shell.

**How to Install the JQ command**

The JQ command is not available in some Linux distributions by default; It needs to be downloaded into the system before it can be used on the terminal; You can download the JQ command just like any other package on your system. On Ubuntu 20.04 use the below-given command to install the JQ utility:

Just replace apt with the package manager of your system if you are running a distribution other than Ubuntu.

If you are running a distribution like CentOS 8 which already has JQ by default then you will get an output similar to this:

**Syntax**

Now we can start using the JQ command as it has been successfully installed on our system, but first, let’s take a look at the syntax of the JQ command:

jq [options] [file...]

jq [options] --args [strings...]

jq [options] --jsonargs [JSON\_TEXTS...]

The JQ command can be used in many different ways; It can be used directly on a JSON file and can also be combined with several other commands to interpret JSON data. The JQ command can be used with different filters such as the **“.”, “|”, “,”** or the **“.[]”** filter to organize JSON data.

The JQ command also takes different options as arguments such as the **--tab**, **--stream**, **--indent n**, **--unbuffered**, and the **-L** directory option. The syntax of the JQ command might seem complex at first but you will get familiar with it once you read the whole article.

**How to Organize JSON data using JQ command**

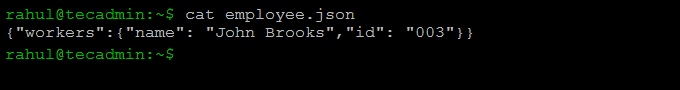
The simplest and frequently used feature of the JQ command filters. They are used to organize and prettify JSON data when printing it to standard output.

In this example, we have a JSON file named employee.json and we need to output the data to the standard output:

{"workers":{"name": "John Brooks","id": "003"}}

We can use the cat command to show the data:

cat employee.json

[](https://tecadmin.net/wp-content/uploads/2021/08/cat-command-example.png)

The data printed to the standard output using the cat command is unorganized and messy. We can organize this data by using the JQ command along with the ‘.’ filter:

[Text

Description automatically generated](https://tecadmin.net/wp-content/uploads/2021/08/jq-example-1.png)

Now the data has become a lot more organized, colorful, and easier to understand. This filter is especially needed when accessing data from APIs; The data stored in APIs can be very unorganized and confusing.

**How to Access a Property using JQ command**

The .field filter along with the JQ command can be used to access object property in the shell.

If we only want to access and print a single property to the standard output then we can use the .field operator. E.g to access the worker’s property we can use this command:

[Text

Description automatically generated](https://tecadmin.net/wp-content/uploads/2021/08/jq-example-2.png)

We can also access the items present within the property by using the .field operator. To access the name item in the worker’s property we will use:

[Text

Description automatically generated](https://tecadmin.net/wp-content/uploads/2021/08/jq-command-example-3.png)

**How to Access an Array Item using JQ command**

We can also access and output the elements present within an array in a JSON file by using the **.[]** operator. For this example we are going to modify our JSON file so it looks like this:

[{"name": "John Brooks","id": "003"},{"name": "Randy Park","id": "053"},{"name": "Todd Gray","id": "009"}]

To output all the arrays present in the JSON file we will run the command given below:

[Text

Description automatically generated](https://tecadmin.net/wp-content/uploads/2021/08/jq-command-example-4.png)

To output only the second array we can modify the above-given command in the following way:

[Text

Description automatically generated](https://tecadmin.net/wp-content/uploads/2021/08/jq-command-example-5.png)

Remember that the array starts at **0**

We can also access the properties present within the array by using the .field operator. E.g if we want to access the name property in the third array then we will run the following command:

[Graphical user interface, text

Description automatically generated](https://tecadmin.net/wp-content/uploads/2021/08/jq-command-example-6.png)

Similarly, to access all the name properties inside arrays we can execute this command:

[Text

Description automatically generated with medium confidence](https://tecadmin.net/wp-content/uploads/2021/08/jq-command-example-7.png)

**Conclusion**

The JQ command is used to transform JSON data into a more readable format and print it to the standard output on Linux. The JQ command is built around filters which are used to find and print only the required data from a JSON file.

In this how-to guide, we have learned to use the JQ command to organize and filter JSON data.

# jq Cheat SheetEdit Cheat Sheet

Example-wise the jq manpage is not really helpful. Let’s document some simple examples here…

To test queries live use [jqplay.org](https://jqplay.org/)

## Output Formatting

If you do only care about output formatting (pretty print) run

jq . my.json

or when reading from a pipeline

cat my.json | jq

Note: for redirection you need to pass a filter too to avoid a syntax error:

jq . my.json > output.json

## jq Extraction Examples

Consider this example document

{

"timestamp": 1234567890,

"report": "Age Report",

"results": [

{ "name": "John", "age": 43, "city": "TownA" },

{ "name": "Joe", "age": 10, "city": "TownB" }

]

}

To extract top level attributes “timestamp” and “report”

jq '. | {timestamp,report}'

To extract name and age of each “results” item

jq '.results[] | {name, age}'

To extract name and age as text values instead of JSON

jq -r '.results[] | {name, age} | join(" ")'

Filter this by attribute

jq '.results[] | select(.name == "John") | {age}' # Get age for 'John'

jq '.results[] | select((.name == "Joe") and (.age = 10))' # Get complete records for all 'Joe' aged 10

jq '.results[] | select(.name | contains("Jo"))' # Get complete records for all names with 'Jo'

jq '.results[] | select(.name | test("Joe\s+Smith"))' # Get complete records for all names matching PCRE regex 'Joe\+Smith'

Avoid null output when accessing non-existing keys

jq '.mykey | select(. != null)'

## “Deep” Value Extraction

If you want to combine subkeys at different levels it won’t work like this

jq '.items[] | { metadata["created"], name }'

Instead you can access values like this

jq '.items[] | { "created" : .metadata["created"], name }'

Or like this

jq '.items[] | .metadata["created"], .name'

The drawback being, that you do not get a JSON output, but each value on a new line.

## Accessing unknown keys

When processing objects you might not know about some keys, in this case use to\_entries. For example if you want to have all property fields of the following JSON:

echo '{

"name": "R1",

"type": "robot",

"prop1": "a5482na",

"prop2": null,

"prop3": 55

}' |\

jq '. | to\_entries[] | select( .key | contains("prop"))'

will give you

{

"key": "prop1",

"value": "a5482na"

}

{

"key": "prop2",

"value": null

}

{

"key": "prop3",

"value": 55

}

## Changing values with jq

Merging/overwriting keys

echo '{ "a": 1, "b": 2 }' |\

jq '. |= . + {

"c": 3

}'

Adding elements to lists

echo '{ "names": ["Marie", "Sophie"] }' |\

jq '.names |= .+ [

"Natalie"

]'

## Delete values with jq

jq 'del(.somekey)' input.json

## Merge JSON strings

For example merge three object lists:

echo '[ {"a":1}, {"b":2} ]' | \

jq --argjson input1 '{ "c":3 }' \

--argjson input2 '[ { "d":4 }, { "e": 5} ]' \

'. = $input1 + . + $input2'

## Merge files (since jq 1.4)

The following command will merge “somekey” from both passed files

jq -s '.[0] \* .[1] | {somekey: .somekey}' <file1> <file2>

## Handle Empty Arrays

When you want to iterate over an array, and the array you access is empty you get something like

jq: error (at <stdin>:3): Cannot iterate over null (null)

To workaround the optional array protect the access with

select(.my\_array | length > 0)

## Testing Types

$ echo '[true, null, 42, "hello", []]' | ./jq 'map(type)'

["boolean","null","number","string","array"]

## Extracting key names

Given an JSON object like this

{

"animals": [

"dog": { },

"cat": { }

]

}

you can extract the names of the animals using

jq '.animals | keys'

## Using jq in Shell Scripts

From <https://www.terraform.io/docs/providers/external/data_source.html>

### **Parsing JSON into env vars**

To fill environment variables from JSON object keys (e.g. $FOO from jq query “.foo”)

export $(jq -r '@sh "FOO=\(.foo) BAZ=\(.baz)"')

To make a bash array

read -a bash\_array < <(jq -r .|arrays|select(.!=null)|@tsv)

### **JSON template using env vars**

To create proper JSON from a shell script and properly escape variables:

jq -n --arg foobaz "$FOOBAZ" '{"foobaz":$foobaz}'

### **URL Encode**

Quick easy way to url encode something

date | jq -sRr @uri

### **String Format / Concat**

There is no printf like function, just concatenation like this:

echo '{ "object" : { "name": "banana", "color": "yellow" }}' |\

jq -r '.object | (.name)+" is "+(.color)'

will print banana is yellow.