
Nextflow Documentation

Release 0.30.2

Paolo Di Tommaso

Jul 12, 2018

Contents

| | | |
|----------|----------------------------------|-----------|
| 1 | Get started | 3 |
| 1.1 | Requirements | 3 |
| 1.2 | Installation | 3 |
| 1.3 | Your first script | 3 |
| 2 | Basic concepts | 7 |
| 2.1 | Processes and channels | 7 |
| 2.2 | Execution abstraction | 8 |
| 2.3 | Scripting language | 9 |
| 2.4 | Configuration options | 9 |
| 3 | Pipeline script | 11 |
| 3.1 | Language basics | 11 |
| 3.2 | Closures | 14 |
| 3.3 | Regular expressions | 15 |
| 3.4 | Files and I/O | 17 |
| 4 | Processes | 27 |
| 4.1 | Script | 28 |
| 4.2 | Inputs | 32 |
| 4.3 | Outputs | 39 |
| 4.4 | When | 44 |
| 4.5 | Directives | 44 |
| 5 | Channels | 63 |
| 5.1 | Channel factory | 63 |
| 5.2 | Binding values | 67 |
| 5.3 | Observing events | 68 |
| 6 | Operators | 71 |
| 6.1 | Filtering operators | 71 |
| 6.2 | Transforming operators | 75 |
| 6.3 | Splitting operators | 83 |
| 6.4 | Combining operators | 89 |
| 6.5 | Forking operators | 98 |
| 6.6 | Maths operators | 101 |
| 6.7 | Other operators | 104 |

| | | |
|-----------|------------------------------------|------------|
| 7 | Executors | 109 |
| 7.1 | Local | 109 |
| 7.2 | SGE | 109 |
| 7.3 | LSF | 110 |
| 7.4 | SLURM | 110 |
| 7.5 | PBS/Torque | 111 |
| 7.6 | NQSII | 111 |
| 7.7 | HTCondor | 112 |
| 7.8 | Ignite | 112 |
| 7.9 | Kubernetes | 112 |
| 7.10 | AWS Batch | 113 |
| 8 | Configuration | 115 |
| 8.1 | Configuration file | 115 |
| 8.2 | Config scopes | 116 |
| 8.3 | Config profiles | 128 |
| 8.4 | Environment variables | 128 |
| 9 | Amazon Cloud | 131 |
| 9.1 | Configuration | 131 |
| 9.2 | Cluster deployment | 133 |
| 9.3 | Pipeline execution | 133 |
| 9.4 | Cluster shutdown | 134 |
| 9.5 | Cluster auto-scaling | 134 |
| 9.6 | Spot prices | 135 |
| 9.7 | Advanced configuration | 135 |
| 9.8 | AWS Batch | 136 |
| 10 | Amazon S3 storage | 139 |
| 10.1 | S3 path | 139 |
| 10.2 | Security credentials | 139 |
| 10.3 | Advanced configuration | 140 |
| 11 | Conda environments | 141 |
| 11.1 | Prerequisites | 141 |
| 11.2 | How it works | 141 |
| 11.3 | Advanced settings | 143 |
| 12 | Docker containers | 145 |
| 12.1 | Prerequisites | 145 |
| 12.2 | How it works | 145 |
| 12.3 | Multiple containers | 146 |
| 12.4 | Executable containers | 147 |
| 12.5 | Advanced settings | 147 |
| 13 | Singularity containers | 149 |
| 13.1 | Prerequisites | 149 |
| 13.2 | Images | 149 |
| 13.3 | How it works | 150 |
| 13.4 | Multiple containers | 150 |
| 13.5 | Singularity & Docker Hub | 151 |
| 13.6 | Advanced settings | 152 |
| 14 | Apache Ignite | 153 |
| 14.1 | Cluster daemon | 153 |

| | | |
|-----------|--|------------|
| 14.2 | Pipeline execution | 155 |
| 14.3 | Execution with MPI | 156 |
| 15 | Kubernetes | 159 |
| 15.1 | Concepts | 159 |
| 15.2 | Requirements | 160 |
| 15.3 | Execution | 160 |
| 15.4 | Interactive login | 161 |
| 15.5 | Running in a pod | 161 |
| 15.6 | Pod settings | 162 |
| 15.7 | Limitation | 162 |
| 15.8 | Advanced configuration | 162 |
| 16 | Tracing & visualisation | 163 |
| 16.1 | Execution report | 163 |
| 16.2 | Trace report | 166 |
| 16.3 | Timeline report | 169 |
| 16.4 | DAG visualisation | 169 |
| 17 | Pipeline sharing | 173 |
| 17.1 | How it works | 173 |
| 17.2 | Running a pipeline | 173 |
| 17.3 | Handling revisions | 174 |
| 17.4 | Commands to manage projects | 174 |
| 17.5 | SCM configuration file | 176 |
| 17.6 | Private server configuration | 177 |
| 17.7 | Local repository configuration | 178 |
| 17.8 | Publishing your pipeline | 178 |
| 17.9 | Manage dependencies | 179 |
| 18 | Workflow introspection | 183 |
| 18.1 | Runtime metadata | 183 |
| 18.2 | Nextflow metadata | 184 |
| 18.3 | Completion handler | 185 |
| 18.4 | Error handler | 185 |
| 18.5 | Notification message | 185 |
| 18.6 | Decoupling metadata | 186 |
| 19 | Mail & Notifications | 187 |
| 19.1 | Mail message | 187 |
| 19.2 | Mail notification | 189 |
| 19.3 | Workflow notification | 190 |
| 20 | Examples | 193 |
| 20.1 | Basic pipeline | 193 |
| 20.2 | More examples | 194 |
| 21 | FAQ | 195 |
| 21.1 | How do I process multiple input files in parallel? | 195 |
| 21.2 | How do I get a unique ID based on the file name? | 196 |
| 21.3 | How do I use the same channel multiple times? | 196 |
| 21.4 | How do I invoke custom scripts and tools? | 197 |
| 21.5 | How do I iterate over a process n times? | 198 |
| 21.6 | How do I iterate over nth files from within a process? | 198 |
| 21.7 | How do I use a specific version of Nextflow? | 199 |

Contents:

1.1 Requirements

Nextflow can be used on any POSIX compatible system (Linux, OS X, etc). It requires BASH and [Java 8](#) (or higher) to be installed.

Windows systems may be supported using a POSIX compatibility layer like [Cygwin](#) (unverified) or, alternatively, installing it into a Linux VM using virtualization software like [VirtualBox](#) or [VMware](#).

1.2 Installation

Nextflow is distributed as a self-contained executable package, which means that it does not require any special installation procedure.

It only needs two easy steps:

1. Download the executable package by copying and pasting the following command in your terminal window:
`wget -qO- https://get.nextflow.io | bash`. It will create the `nextflow` main executable file in the current directory.
2. Optionally, move the `nextflow` file to a directory accessible by your `$PATH` variable (this is only required to avoid remembering and typing the full path to `nextflow` each time you need to run it).

Tip: In the case you don't have `wget` installed you can use the `curl` utility instead by entering the following command: `curl -s https://get.nextflow.io | bash`

1.3 Your first script

Copy the following example into your favourite text editor and save it to a file named `tutorial.nf`

```
#!/usr/bin/env nextflow

params.str = 'Hello world!'

process splitLetters {

    output:
    file 'chunk_*' into letters mode flatten

    """
    printf '${params.str}' | split -b 6 - chunk_
    """
}

process convertToUpper {

    input:
    file x from letters

    output:
    stdout result

    """
    cat $x | tr '[a-z]' '[A-Z]'
    """
}

result.subscribe {
    println it.trim()
}
```

This script defines two processes. The first splits a string in file chunks containing 6 characters, and the second receives these files and transforms their contents to uppercase letters. The resulting strings are emitted on the `result` channel and the final output is printed by the `subscribe` operator.

Execute the script by entering the following command in your terminal:

```
nextflow run tutorial.nf
```

It will output something similar to the text shown below:

```
N E X T F L O W ~ version 0.9.0
[warm up] executor > local
[22/7548fa] Submitted process > splitLetters (1)
[e2/008ee9] Submitted process > convertToUpper (1)
[1e/165130] Submitted process > convertToUpper (2)
HELLO
WORLD!
```

You can see that the first process is executed once, and the second twice. Finally the result string is printed.

It's worth noting that the process `convertToUpper` is executed in parallel, so there's no guarantee that the instance processing the first split (the chunk *Hello*) will be executed before before the one processing the second split (the chunk *world!*).

Thus, it is perfectly possible that you will get the final result printed out in a different order:

```
WORLD!
HELLO
```

Tip: The hexadecimal numbers, like `22/7548fa`, identify the unique process execution. These numbers are also the prefix of the directories where each process is executed. You can inspect the files produced by them changing to the directory `$PWD/work` and using these numbers to find the process-specific execution path.

1.3.1 Modify and resume

Nextflow keeps track of all the processes executed in your pipeline. If you modify some parts of your script, only the processes that are actually changed will be re-executed. The execution of the processes that are not changed will be skipped and the cached result used instead.

This helps a lot when testing or modifying part of your pipeline without having to re-execute it from scratch.

For the sake of this tutorial, modify the `convertToUpper` process in the previous example, replacing the process script with the string `rev $x`, so that the process looks like this:

```
process convertToUpper {

    input:
    file x from letters

    output:
    stdout result

    """
    rev $x
    """
}
```

Then save the file with the same name, and execute it by adding the `-resume` option to the command line:

```
nextflow run tutorial.nf -resume
```

It will print output similar to this:

```
N E X T F L O W ~ version 0.9.0
[warm up] executor > local
[22/7548fa] Cached process > splitLetters (1)
[d0/7b79a3] Submitted process > convertToUpper (1)
[b0/c99ef9] Submitted process > convertToUpper (2)
olleH
!dlrow
```

You will see that the execution of the process `splitLetters` is actually skipped (the process ID is the same), and its results are retrieved from the cache. The second process is executed as expected, printing the reversed strings.

Tip: The pipeline results are cached by default in the directory `$PWD/work`. Depending on your script, this folder can take of lot of disk space. If your are sure you won't resume your pipeline execution, clean this folder periodically.

1.3.2 Pipeline parameters

Pipeline parameters are simply declared by prepending to a variable name the prefix `params`, separated by dot character. Their value can be specified on the command line by prefixing the parameter name with a double *dash* character, i.e. `--paramName`

For the sake of this tutorial, you can try to execute the previous example specifying a different input string parameter, as shown below:

```
nextflow run tutorial.nf --str 'Hola mundo'
```

The string specified on the command line will override the default value of the parameter. The output will look like this:

```
N E X T F L O W ~ version 0.7.0
[warm up] executor > local
[6d/54ab39] Submitted process > splitLetters (1)
[a1/88716d] Submitted process > convertToUpper (2)
[7d/3561b6] Submitted process > convertToUpper (1)
odnu
m aloH
```

CHAPTER 2

Basic concepts

Nextflow is a reactive workflow framework and a programming [DSL](#) that eases writing computational pipelines with complex data.

It is designed around the idea that the Linux platform is the lingua franca of data science. Linux provides many simple but powerful command-line and scripting tools that, when chained together, facilitate complex data manipulations.

Nextflow extends this approach, adding the ability to define complex program interactions and a high-level parallel computational environment based on the *dataflow* programming model.

2.1 Processes and channels

In practice a *Nextflow* pipeline script is made by joining together many different processes. Each process can be written in any scripting language that can be executed by the Linux platform (Bash, Perl, Ruby, Python, etc.).

Processes are executed independently and are isolated from each other, i.e. they do not share a common (writable) state. The only way they can communicate is via asynchronous FIFO queues, called *channels* in the *Nextflow* lingo.

Any process can define one or more channels as *input* and *output*. The interaction between these processes, and ultimately the pipeline execution flow itself, is implicitly defined by these input and output declarations.

A *Nextflow* script looks like this:

```
params.query = "$HOME/sample.fa"
params.db = "$HOME/tools/blast-db/pdb/pdb"

db = file(params.db)
query = file(params.query)

process blastSearch {
    input:
        file query

    output:
```

(continues on next page)

(continued from previous page)

```
file top_hits

"""
blastp -db $db -query $query -outfmt 6 > blast_result
cat blast_result | head -n 10 | cut -f 2 > top_hits
"""
}

process extractTopHits {
  input:
    file top_hits

  output:
    file sequences

  """
  blastdbcmd -db ${db} -entry_batch $top_hits > sequences
  """
}
```

The above example defines two processes. Their execution order is not determined by the fact that the `blastSearch` process comes before the `extractTopHits` in the script (it could also be written the other way around), but because the first defines the channel `top_hits` in its output declarations while the `extractTopHits` process defines the same channel in its input declaration, thus establishing a communication link from the *blastSearch* task towards the *extractTopHits* task.

Read the [Channel](#) and [Process](#) sections to learn more about these features.

2.2 Execution abstraction

While a process defines *what* command or script has to be executed, the *executor* determines *how* that script is actually run on the target system.

If not otherwise specified, processes are executed on the local computer. The local executor is very useful for pipeline development and test purposes, but for real world computational pipelines an HPC or cloud platform is required.

In other words, *Nextflow* provides an abstraction between the pipeline's functional logic and the underlying execution system. Thus it is possible to write a pipeline once and to seamlessly run it on your computer, a grid platform, or the cloud, without modifying it, by simply defining the target execution platform in the configuration file.

The following HPC and cloud platforms are supported:

- Open grid engine
- Univa grid engine
- Platform LSF
- Linux SLURM
- PBS Works
- Torque
- HTCondor

Read the [Executors](#) section to learn more about Nextflow executors.

2.3 Scripting language

Although *Nextflow* is designed to be used with a minimal learning curve, without having to study a new programming language and using your current skills, it also provides a powerful scripting DSL.

Nextflow scripting is an extension of the [Groovy programming language](#), which in turn is a super-set of the Java programming language. Thus if you have some knowledge of these languages, or even just some confidence with the C/C++ syntax, you will be comfortable using it.

Read the [Pipeline script](#) section to learn about the Nextflow scripting language.

2.4 Configuration options

Pipeline configuration properties are defined in a file named `nextflow.config` in the pipeline execution directory.

This file can be used to define which executor to use, the process's environment variables, pipeline parameters etc.

A basic configuration file might look like this:

```
process {
  executor='sge'
  queue = 'cn-el6'
}

env {
  PATH="$PWD/bowtie2:$PWD/tophat2:$PATH"
}
```

Read the [Configuration](#) section to learn more about the Nextflow configuration file and settings.

CHAPTER 3

Pipeline script

The Nextflow scripting language is an extension of the Groovy programming language whose syntax has been specialized to ease the writing of computational pipelines in a declarative manner.

This means that Nextflow can execute any piece of Groovy code or use any library for the JVM platform.

For a detailed description of the Groovy programming language, reference these links:

- [Groovy User Guide](#)
- [Groovy Cheat sheet](#)
- [Groovy in Action](#)

Below you can find a crash course in the most important language constructs used in the Nextflow scripting language.

Warning: Nextflow uses UTF-8 as the default file character encoding for source and application files. Make sure to use the UTF-8 encoding when editing Nextflow scripts with your favourite text editor.

3.1 Language basics

3.1.1 Hello world

To print something is as easy as using one of the `print` or `println` methods.

```
println "Hello, World!"
```

The only difference between the two is that the `println` method implicitly appends a *new line* character to the printed string.

3.1.2 Variables

To define a variable, simply assign a value to it:

```
x = 1
println x

x = new java.util.Date()
println x

x = -3.1499392
println x

x = false
println x

x = "Hi"
println x
```

3.1.3 Lists

A List object can be defined by placing the list items in square brackets:

```
myList = [1776, -1, 33, 99, 0, 928734928763]
```

You can access a given item in the list with square-bracket notation (indexes start at 0):

```
println myList[0]
```

In order to get the length of the list use the `size` method:

```
println myList.size()
```

Learn more about lists:

- [Groovy Lists tutorial](#)
- [Groovy List SDK](#)
- [Java List SDK](#)

3.1.4 Maps

Maps are used to store *associative arrays* or *dictionaries*. They are unordered collections of heterogeneous, named data:

```
scores = [ "Brett":100, "Pete":"Did not finish", "Andrew":86.87934 ]
```

Note that each of the values stored in the map can be of a different type. Brett is an integer, Pete is a string, and Andrew is a floating-point number.

We can access the values in a map in two main ways:

```
println scores["Pete"]
println scores.Pete
```

To add data to or modify a map, the syntax is similar to adding values to list:

```
scores["Pete"] = 3
scores["Cedric"] = 120
```

Learn more about maps:

- [Groovy Maps tutorial](#)
- [Groovy Map SDK](#)
- [Java Map SDK](#)

3.1.5 Multiple assignment

An array or a list object can be used to assign to multiple variables at once:

```
(a, b, c) = [10, 20, 'foo']
assert a == 10 && b == 20 && c == 'foo'
```

The three variables on the left of the assignment operator are initialized by the corresponding item in the list.

Read more about [Multiple assignment](#) in the Groovy documentation.

3.1.6 Conditional Execution

One of the most important features of any programming language is the ability to execute different code under different conditions. The simplest way to do this is to use the `if` construct:

```
x = Math.random()
if( x < 0.5 ) {
    println "You lost."
}
else {
    println "You won!"
}
```

3.1.7 Strings

Strings can be defined by enclosing text in single or double quotes (`'` or `"` characters):

```
println "he said 'cheese' once"
println 'he said "cheese!" again'
```

Strings can be concatenated with `+`:

```
a = "world"
print "hello " + a + "\n"
```

3.1.8 String interpolation

There is an important difference between single- and double-quoted strings: Double-quoted strings support variable interpolations, while single-quoted strings do not.

In practice, double-quoted strings can contain the value of an arbitrary variable by prefixing its name with the `$` character, or the value of any expression by using the `${expression}` syntax, similar to Bash/shell scripts:

```
foxttype = 'quick'
foxcolor = ['b', 'r', 'o', 'w', 'n']
println "The $foxttype ${foxcolor.join()} fox"

x = 'Hello'
println '$x + $y'
```

This code prints:

```
The quick brown fox
$x + $y
```

3.1.9 Multi-line strings

A block of text that span multiple lines can be defined by delimiting it with triple single or double quotes:

```
text = """
    hello there James
    how are you today?
    """
```

Note: Like before, multi-line strings inside double quotes support variable interpolation, while single-quoted multi-line strings do not.

As in Bash/shell scripts, terminating a line in a multi-line string with a `\` character prevents a *new line* character from separating that line from the one that follows:

```
myLongCmdline = """ blastp \
    -in $input_query \
    -out $output_file \
    -db $blast_database \
    -html
    """

result = myLongCmdline.execute().text
```

In the preceding example, `blastp` and its `-in`, `-out`, `-db` and `-html` switches and their arguments are effectively a single line.

3.2 Closures

Briefly, a closure is a block of code that can be passed as an argument to a function. Thus, you can define a chunk of code and then pass it around as if it were a string or an integer.

More formally, you can create functions that are defined as *first class objects*.

```
square = { it * it }
```

The curly brackets around the expression `it * it` tells the script interpreter to treat this expression as code. In this case, the designator `it` refers to whatever value is passed to the function when it is called. This compiled function is assigned to the variable `square`, much like variable assignments shown previously. Now we can do something like this:

```
println square(9)
```

and get the value 81.

This is not very interesting until we find that we can pass the function `square` as an argument to other functions or methods. Some built-in functions take a function like this as an argument. One example is the `collect` method on lists:

```
[ 1, 2, 3, 4 ].collect(square)
```

This expression says: Create an array with the values 1, 2, 3 and 4, then call its `collect` method, passing in the closure we defined above. The `collect` method runs through each item in the array, calls the closure on the item, then puts the result in a new array, resulting in:

```
[ 1, 4, 9, 16 ]
```

For more methods that you can call with closures as arguments, see the [Groovy GDK documentation](#).

By default, closures take a single parameter called `it`, but you can also create closures with multiple, custom-named parameters. For example, the method `Map.each()` can take a closure with two arguments, to which it binds the *key* and the associated *value* for each key-value pair in the `Map`. Here, we use the obvious variable names `key` and `value` in our closure:

```
printMapClosure = { key, value ->
    println "$key = $value"
}

[ "Yue" : "Wu", "Mark" : "Williams", "sudha" : "Kumari" ].each(printMapClosure)
```

Prints:

```
Yue=Wu
Mark=Williams
Sudha=Kumari
```

A closure has two other important features. First, it can access variables in the scope where it is defined, so that it can *interact* with them.

Second, a closure can be defined in an *anonymous* manner, meaning that it is not given a name, and is defined in the place where it needs to be used.

As an example showing both these features, see the following code fragment:

```
myMap = ["China": 1, "India" : 2, "USA" : 3]

result = 0
myMap.keySet().each( { result+= myMap[it] } )

println result
```

Learn more about closures in the [Groovy documentation](#)

3.3 Regular expressions

Regular expressions are the Swiss Army knife of text processing. They provide the programmer with the ability to match and extract patterns from strings.

Regular expressions are available via the `~/pattern/` syntax and the `==~` and `==~` operators.

Use `==~` to check whether a given pattern occurs anywhere in a string:

```
assert 'foo' ==~ /foo/      // return TRUE
assert 'foobar' ==~ /foo/   // return TRUE
```

Use `==~` to check whether a string matches a given regular expression pattern exactly.

```
assert 'foo' ==~ /foo/      // return TRUE
assert 'foobar' ==~ /foo/   // return FALSE
```

It is worth noting that the `~/` operator creates a Java `Pattern` object from the given string, while the `==~` operator creates a Java `Matcher` object.

```
x = ~/abc/
println x.class
// prints java.util.regex.Pattern

y = 'some string' ==~ /abc/
println y.class
// prints java.util.regex.Matcher
```

Regular expression support is imported from Java. Java's regular expression language and API is documented in the [Pattern Java documentation](#).

You may also be interested in this post: [Groovy: Don't Fear the RegExp](#).

3.3.1 String replacement

To replace pattern occurrences in a given string, use the `replaceFirst` and `replaceAll` methods:

```
x = "colour".replaceFirst(/ou/, "o")
println x
// prints: color

y = "cheesecheese".replaceAll(/cheese/, "nice")
println y
// prints: nicenice
```

3.3.2 Capturing groups

You can match a pattern that includes groups. First create a matcher object with the `==~` operator. Then, you can index the matcher object to find the matches: `matcher[0]` returns a list representing the first match of the regular expression in the string. The first list element is the string that matches the entire regular expression, and the remaining elements are the strings that match each group.

Here's how it works:

```
programVersion = '2.7.3-beta'
m = programVersion ==~ /(\d+)\.(\d+)\.(\d+)-?(.+)/

assert m[0] == ['2.7.3-beta', '2', '7', '3', 'beta']
assert m[0][1] == '2'
assert m[0][2] == '7'
```

(continues on next page)

(continued from previous page)

```
assert m[0][3] == '3'
assert m[0][4] == 'beta'
```

Applying some syntactic sugar, you can do the same in just one line of code:

```
programVersion = '2.7.3-beta'
(full, major, minor, patch, flavor) = (programVersion =~ /(\d+)\.(\d+)\.(\d+)-?(.+)/
↪)[0]

println full      // 2.7.3-beta
println major     // 2
println minor     // 7
println patch     // 3
println flavor    // beta
```

3.3.3 Removing part of a string

You can remove part of a `String` value using a regular expression pattern. The first match found is replaced with an empty `String`:

```
// define the regexp pattern
wordStartsWithGr = ~/(?i)\s+Gr\w+/

// apply and verify the result
('Hello Groovy world!' - wordStartsWithGr) == 'Hello world!'
('Hi Grails users' - wordStartsWithGr) == 'Hi users'
```

Remove the first 5-character word from a string:

```
assert ('Remove first match of 5 letter word' - ~/\b\w{5}\b/) == 'Remove match of 5_
↪letter word'
```

Remove the first number with its trailing whitespace from a string:

```
assert ('Line contains 20 characters' - ~/\d+\s+/) == 'Line contains characters'
```

3.4 Files and I/O

To access and work with files, use the `file` method, which returns a file system object given a file path string:

```
myFile = file('some/path/to/my_file.file')
```

The `file` method can reference either *files* or *directories*, depending on what the string path refers to in the file system.

When using the wildcard characters `*`, `?`, `[]` and `{ }`, the argument is interpreted as a `glob` path matcher and the `file` method returns a list object holding the paths of files whose names match the specified pattern, or an empty list if no match is found:

```
listOfFiles = file('some/path/*.fa')
```

Note: Two asterisks (`**`) in a glob pattern works like `*` but matches any number of directory components in a file system path.

By default, wildcard characters do not match directories or hidden files. For example, if you want to include hidden files in the result list, add the optional parameter `hidden`:

```
listWithHidden = file('some/path/*.fa', hidden: true)
```

Here are `file`'s available options:

| Name | Description |
|--------------------------|--|
| <code>glob</code> | When <code>true</code> interprets characters <code>*</code> , <code>?</code> , <code>[]</code> and <code>{ }</code> as glob wildcards, otherwise handles them as normal characters (default: <code>true</code>) |
| <code>type</code> | Type of paths returned, either <code>file</code> , <code>dir</code> or <code>any</code> (default: <code>file</code>) |
| <code>hidden</code> | When <code>true</code> includes hidden files in the resulting paths (default: <code>false</code>) |
| <code>maxDepth</code> | Maximum number of directory levels to visit (default: <i>no limit</i>) |
| <code>followLinks</code> | When <code>true</code> follows symbolic links during directory tree traversal, otherwise treats them as files (default: <code>true</code>) |

Tip: If you are a Java geek you will be interested to know that the `file` method returns a `Path` object, which allows you to use the usual methods you would in a Java program.

See also: [Channel.fromPath](#) .

3.4.1 Basic read/write

Given a file variable, declared using the `file` method as shown in the previous example, reading a file is as easy as getting the value of the file's `text` property, which returns the file content as a string value:

```
print myFile.text
```

Similarly, you can save a string value to a file by simply assigning it to the file's `text` property:

```
myFile.text = 'Hello world!'
```

Note: Existing file content is overwritten by the assignment operation, which also implicitly creates files that do not exist.

In order to append a string value to a file without erasing existing content, you can use the `append` method:

```
myFile.append('Add this line\n')
```

Or use the *left shift* operator, a more idiomatic way to append text content to a file:

```
myFile << 'Add a line more\n'
```

Binary data can managed in the same way, just using the file property `bytes` instead of `text`. Thus, the following example reads the file and returns its content as a byte array:


```
binaryContent = myFile.bytes
```

Or you can save a byte array data buffer to a file, by simply writing:

```
myFile.bytes = binaryBuffer
```

Warning: The above methods read and write ALL the file content at once, in a single variable or buffer. For this reason they are not suggested when dealing with big files, which require a more memory efficient approach, for example reading a file line by line or by using a fixed size buffer.

3.4.2 Read a file line by line

In order to read a text file line by line you can use the method `readLines()` provided by the file object, which returns the file content as a list of strings:

```
myFile = file('some/my_file.txt')
allLines = myFile.readLines()
for( line : allLines ) {
    println line
}
```

This can also be written in a more idiomatic syntax:

```
file('some/my_file.txt')
    .readLines()
    .each { println it }
```

Note: The method `readLines()` reads all the file content at once and returns a list containing all the lines. For this reason, do not use it to read big files.

To process a big file, use the method `eachLine`, which reads only a single line at a time into memory:

```
count = 0
myFile.eachLine { str ->
    println "line ${count++}: $str"
}
```

3.4.3 Advanced file reading operations

The classes `Reader` and `InputStream` provide fine control for reading text and binary files, respectively.

The method `newReader` creates a `Reader` object for the given file that allows you to read the content as single characters, lines or arrays of characters:

```
myReader = myFile.newReader()
String line
while( line = myReader.readLine() ) {
    println line
}
myReader.close()
```

The method `withReader` works similarly, but automatically calls the `close` method for you when you have finished processing the file. So, the previous example can be written more simply as:

```
myFile.withReader {
    String line
    while( line = myReader.readLine() ) {
        println line
    }
}
```

The methods `newInputStream` and `withInputStream` work similarly. The main difference is that they create an `InputStream` object useful for writing binary data.

Here are the most important methods for reading from files:

| Name | Description |
|------------------------------|---|
| <code>getText</code> | Returns the file content as a string value |
| <code>getBytes</code> | Returns the file content as byte array |
| <code>readLines</code> | Reads the file line by line and returns the content as a list of strings |
| <code>eachLine</code> | Iterates over the file line by line, applying the specified <i>closure</i> |
| <code>eachByte</code> | Iterates over the file byte by byte, applying the specified <i>closure</i> |
| <code>withReader</code> | Opens a file for reading and lets you access it with a <code>Reader</code> object |
| <code>withInputStream</code> | Opens a file for reading and lets you access it with an <code>InputStream</code> object |
| <code>newReader</code> | Returns a <code>Reader</code> object to read a text file |
| <code>newInputStream</code> | Returns an <code>InputStream</code> object to read a binary file |

Read the Java documentation for `Reader` and `InputStream` classes to learn more about methods available for reading data from files.

3.4.4 Advanced file writing operations

The `Writer` and `OutputStream` classes provide fine control for writing text and binary files, respectively, including low-level operations for single characters or bytes, and support for big files.

For example, given two file objects `sourceFile` and `targetFile`, the following code copies the first file's content into the second file, replacing all U characters with X:

```
sourceFile.withReader { source ->
    targetFile.withWriter { target ->
        String line
        while( line=source.readLine() ) {
            target << line.replaceAll('U','X')
        }
    }
}
```

Here are the most important methods for writing to files:

| Name | Description |
|------------------|---|
| setText | Writes a string value to a file |
| setBytes | Writes a byte array to a file |
| write | Writes a string to a file, replacing any existing content |
| append | Appends a string value to a file without replacing existing content |
| newWriter | Creates a Writer object that allows you to save text data to a file |
| newPrintWriter | Creates a PrintWriter object that allows you to write formatted text to a file |
| newOutputStream | Creates an OutputStream object that allows you to write binary data to a file |
| withWriter | Applies the specified closure to a Writer object, closing it when finished |
| withPrintWriter | Applies the specified closure to a PrintWriter object, closing it when finished |
| withOutputStream | Applies the specified closure to an OutputStream object, closing it when finished |

Read the Java documentation for the [Writer](#), [PrintWriter](#) and [OutputStream](#) classes to learn more about methods available for writing data to files.

3.4.5 List directory content

Let's assume that you need to walk through a directory of your choice. You can define the `myDir` variable that points to it:

```
myDir = file('any/path')
```

The simplest way to get a directory list is by using the methods `list` or `listFiles`, which return a collection of first-level elements (files and directories) of a directory:

```
allFiles = myDir.list()
for( def file : allFiles ) {
    println file
}
```

Note: The only difference between `list` and `listFiles` is that the former returns a list of strings, and the latter a list of file objects that allow you to access file metadata, e.g. size, last modified time, etc.

The `eachFile` method allows you to iterate through the first-level elements only (just like `listFiles`). As with other *each-* methods, `eachFiles` takes a closure as a parameter:

```
myDir.eachFile { item ->
    if( item.isFile() ) {
        println "${item.getName()} - size: ${item.size()}"
    }
    else if( item.isDirectory() ) {
        println "${item.getName()} - DIR"
    }
}
```

Several variants of the above method are available. See the table below for a complete list.

| Name | Description |
|------------------------------|---|
| <code>eachFile</code> | Iterates through first-level elements (files and directories). Read more |
| <code>eachDir</code> | Iterates through first-level directories only. Read more |
| <code>eachFileMatch</code> | Iterates through files and dirs whose names match the given filter. Read more |
| <code>eachDirMatch</code> | Iterates through directories whose names match the given filter. Read more |
| <code>eachFileRecurse</code> | Iterates through directory elements depth-first. Read more |
| <code>eachDirRecurse</code> | Iterates through directories depth-first (regular files are ignored). Read more |

See also: Channel *fromPath* method.

3.4.6 Create directories

Given a file variable representing a nonexistent directory, like the following:

```
myDir = file('any/path')
```

the method `mkdir` creates a directory at the given path, returning `true` if the directory is created successfully, and `false` otherwise:

```
result = myDir.mkdir()
println result ? "OK" : "Cannot create directory: $myDir"
```

Note: If the parent directories do not exist, the above method will fail and return `false`.

The method `mkdirs` creates the directory named by the file object, including any nonexistent parent directories:

```
myDir.mkdirs()
```

3.4.7 Create links

Given a file, the method `mlink` creates a *file system link* for that file using the path specified as a parameter:

```
myFile = file('/some/path/file.txt')
myFile.mlink('/user/name/link-to-file.txt')
```

Table of optional parameters:

| Name | Description |
|-------------------------|---|
| <code>hard</code> | When <code>true</code> creates a <i>hard</i> link, otherwise creates a <i>soft</i> (aka <i>symbolic</i>) link. (default: <code>false</code>) |
| <code>over-write</code> | When <code>true</code> overwrites any existing file with the same name, otherwise throws a <code>FileAlreadyExistsException</code> (default: <code>false</code>) |

3.4.8 Copy files

The method `copyTo` copies a file into a new file or into a directory, or copie a directory to a new directory:

```
myFile.copyTo('new_name.txt')
```

Note: If the target file already exists, it will be replaced by the new one. Note also that, if the target is a directory, the source file will be copied into that directory, maintaining the file's original name.

When the source file is a directory, all its content is copied to the target directory:

```
myDir = file('/some/path')
myDir.copyTo('/some/new/path')
```

If the target path does **not** exist, it will be created automatically.

Tip: The `copyTo` method mimics the semantics of the Linux command `cp -r <source> <target>`, with the following caveat: While Linux tools often treat paths ending with a slash (e.g. `/some/path/name/`) as directories, and those not (e.g. `/some/path/name`) as regular files, Nextflow (due to its use of the Java files API) views both these paths as the same file system object. If the path exists, it is handled according to its actual type (i.e. as a regular file or as a directory). If the path does not exist, it is treated as a regular file, with any missing parent directories created automatically.

3.4.9 Move files

You can move a file by using the method `moveTo`:

```
myFile = file('/some/path/file.txt')
myFile.moveTo('/another/path/new_file.txt')
```

Note: When a file with the same name as the target already exists, it will be replaced by the source. Note also that, when the target is a directory, the file will be moved to (or within) that directory, maintaining the file's original name.

When the source is a directory, all the directory content is moved to the target directory:

```
myDir = file('/any/dir_a')
myDir.moveTo('/any/dir_b')
```

Please note that the result of the above example depends on the existence of the target directory. If the target directory exists, the source is moved into the target directory, resulting in the path:

```
/any/dir_b/dir_a
```

If the target directory does not exist, the source is just renamed to the target name, resulting in the path:

```
/any/dir_b
```

Tip: The `moveTo` method mimics the semantics of the Linux command `mv <source> <target>`, with the same caveat as that given for `copyTo`, above.

3.4.10 Rename files

You can rename a file or directory by simply using the `renameTo` file method:

```
myFile = file('my_file.txt')
myFile.renameTo('new_file_name.txt')
```

3.4.11 Delete files

The file method `delete` deletes the file or directory at the given path, returning `true` if the operation succeeds, and `false` otherwise:

```
myFile = file('some/file.txt')
result = myFile.delete
println result ? "OK" : "Can delete: $myFile"
```

Note: This method deletes a directory **ONLY** if it does not contain any files or sub-directories. To delete a directory and **ALL** its content (i.e. removing all the files and sub-directories it may contain), use the method `deleteDir`.

3.4.12 Check file attributes

The following methods can be used on a file variable created by using the `file` method:

| Name | Description |
|---------------------------|--|
| <code>getName</code> | Gets the file name e.g. <code>/some/path/file.txt -> file.txt</code> |
| <code>getBaseName</code> | Gets the file name without its extension e.g. <code>/some/path/file.txt -> file</code> |
| <code>getExtension</code> | Gets the file extension e.g. <code>/some/path/file.txt -> txt</code> |
| <code>getParent</code> | Gets the file parent path e.g. <code>/some/path/file.txt -> /some/path</code> |
| <code>size</code> | Gets the file size in bytes |
| <code>exists</code> | Returns <code>true</code> if the file exists, or <code>false</code> otherwise |
| <code>isEmpty</code> | Returns <code>true</code> if the file is zero length or does not exist, <code>false</code> otherwise |
| <code>isFile</code> | Returns <code>true</code> if it is a regular file e.g. not a directory |
| <code>isDirectory</code> | Returns <code>true</code> if the file is a directory |
| <code>isHidden</code> | Returns <code>true</code> if the file is hidden |
| <code>lastModified</code> | Returns the file last modified timestamp i.e. a long as Linux epoch time |

For example, the following line prints a file name and size:

```
println "File ${myFile.getName()} size: ${myFile.size()}"
```

3.4.13 Get and modify file permissions

Given a file variable representing a file (or directory), the method `getPermissions` returns a 9-character string representing the file's permissions using the [Linux symbolic notation](#) e.g. `rw-rw-r--`:

```
permissions = myFile.getPermissions()
```

Similarly, the method `setPermissions` sets the file's permissions using the same notation:

```
myFile.setPermissions('rwxr-xr-x')
```

A second version of the `setPermissions` method sets a file's permissions given three digits representing, respectively, the *owner*, *group* and *other* permissions:

```
myFile.setPermissions(7,5,5)
```

Learn more about [File permissions numeric notation](#).

3.4.14 HTTP/FTP files

Nextflow provides transparent integration of HTTP/S and FTP protocols for handling remote resources as local file system objects. Simply specify the resource URL as the argument of the *file* object:

```
pdb = file('http://files.rcsb.org/header/5FID.pdb')
```

Then, you can access it as a local file as described in the previous sections:

```
println pdb.text
```

The above one-liner prints the content of the remote PDB file. Previous sections provide code examples showing how to stream or copy the content of files.

Note: Write and list operations are not supported for HTTP/S and FTP files.

CHAPTER 4

Processes

In Nextflow a *process* is the basic processing *primitive* to execute a user script.

The process definition starts with keyword the `process`, followed by process name and finally the process *body* delimited by curly brackets. The process body must contain a string which represents the command or, more generally, a script that is executed by it. A basic process looks like the following example:

```
process sayHello {  
  
    """  
    echo 'Hello world!' > file  
    """  
  
}
```

more specifically a process may contain five definition blocks, respectively: directives, inputs, outputs, when clause and finally the process script. The syntax is defined as follows:

```
process < name > {  
  
    [ directives ]  
  
    input:  
    < process inputs >  
  
    output:  
    < process outputs >  
  
    when:  
    < condition >  
  
    [script|shell|exec]:  
    < user script to be executed >  
  
}
```

4.1 Script

The *script* block is a string statement that defines the command that is executed by the process to carry out its task.

A process contains one and only one script block, and it must be the last statement when the process contains input and output declarations.

The entered string is executed as a **BASH** script in the *host* system. It can be any command, script or combination of them, that you would normally use in terminal shell or in a common BASH script.

The only limitation to the commands that can be used in the script statement is given by the availability of those programs in the target execution system.

The script block can be a simple string or multi-line string. The latter simplifies the writing of non trivial scripts composed by multiple commands spanning over multiple lines. For example:

```
process doMoreThings {  
  
    """  
    blastp -db $db -query query.fa -outfmt 6 > blast_result  
    cat blast_result | head -n 10 | cut -f 2 > top_hits  
    blastdbcmd -db $db -entry_batch top_hits > sequences  
    """  
  
}
```

As explained in the script tutorial section, strings can be defined by using a single-quote or a double-quote, and multi-line strings are defined by three single-quote or three double-quote characters.

There is a subtle but important difference between them. Like in BASH, strings delimited by a " character support variable substitutions, while strings delimited by ' do not.

In the above code fragment the `$db` variable is replaced by the actual value defined somewhere in the pipeline script.

Warning: Since Nextflow uses the same BASH syntax for variable substitutions in strings, you need to manage them carefully depending on if you want to evaluate a variable in the Nextflow context - or - in the BASH environment execution.

When you need to access a system environment variable in your script you have two options. The first choice is as easy as defining your script block by using a single-quote string. For example:

```
process printPath {  
  
    '''  
    echo The path is: $PATH  
    '''  
  
}
```

The drawback of this solution is that you will not be able to access variables defined in the pipeline script context, in your script block.

To fix this, define your script by using a double-quote string and *escape* the system environment variables by prefixing them with a back-slash \ character, as shown in the following example:

```
process doOtherThings {
```

(continues on next page)

(continued from previous page)

```

"""
blastp -db $DB -query query.fa -outfmt 6 > blast_result
cat blast_result | head -n $MAX | cut -f 2 > top_hits
blastdbcmd -db $DB -entry_batch top_hits > sequences
"""
}

```

In this example the `$MAX` variable has to be defined somewhere before, in the pipeline script. *Nextflow* replaces it with the actual value before executing the script. Instead, the `$DB` variable must exist in the script execution environment and the BASH interpreter will replace it with the actual value.

Tip: Alternatively you can use the *Shell* block definition which allows a script to contain both BASH and Nextflow variables without having to escape the first.

4.1.1 Scripts à la carte

The process script is interpreted by Nextflow as a BASH script by default, but you are not limited to it.

You can use your favourite scripting language (e.g. Perl, Python, Ruby, R, etc), or even mix them in the same pipeline.

A pipeline may be composed by processes that execute very different tasks. Using *Nextflow* you can choose the scripting language that better fits the task carried out by a specified process. For example for some processes *R* could be more useful than *Perl*, in other you may need to use *Python* because it provides better access to a library or an API, etc.

To use a scripting other than BASH, simply start your process script with the corresponding *shebang* declaration. For example:

```

process perlStuff {

    """
    #!/usr/bin/perl

    print 'Hi there!' . '\n';
    """
}

process pyStuff {

    """
    #!/usr/bin/python

    x = 'Hello'
    y = 'world!'
    print "%s - %s" % (x,y)
    """
}

```

Tip: Since the actual location of the interpreter binary file can change across platforms, to make your scripts more portable it is wise to use the `env` shell command followed by the interpreter's name, instead of the absolute path of it.

Thus, the *shebang* declaration for a Perl script, for example, would look like: `#!/usr/bin/env perl` instead of the one in the above pipeline fragment.

4.1.2 Conditional scripts

Complex process scripts may need to evaluate conditions on the input parameters or use traditional flow control statements (i.e. `if`, `switch`, etc) in order to execute specific script commands, depending on the current inputs configuration.

Process scripts can contain conditional statements by simply prefixing the script block with the keyword `script:`. By doing that the interpreter will evaluate all the following statements as a code block that must return the script string to be executed. It's much easier to use than to explain, for example:

```
seq_to_align = ...
mode = 'tcoffee'

process align {
    input:
        file seq_to_aln from sequences

    script:
        if( mode == 'tcoffee' )
            """
            t_coffee -in $seq_to_aln > out_file
            """

        else if( mode == 'mafft' )
            """
            mafft --anysymbol --parttree --quiet $seq_to_aln > out_file
            """

        else if( mode == 'clustalo' )
            """
            clustalo -i $seq_to_aln -o out_file
            """

        else
            error "Invalid alignment mode: ${mode}"
}
```

In the above example the process will execute the script fragment depending on the value of the `mode` parameter. By default it will execute the `tcoffee` command, changing the `mode` variable to `mafft` or `clustalo` value, the other branches will be executed.

4.1.3 Template

Process script can be externalised by using *template* files which can be reused across different processes and tested independently by the overall pipeline execution.

A template is simply a shell script file that Nextflow is able to execute by using the `template` function as shown below:

```
process template_example {
    input:
    val STR from 'this', 'that'

    script:
    template 'my_script.sh'
}
```

Nextflow looks for the `my_script.sh` template file in the directory `templates` that must exist in the same folder where the Nextflow script file is located (any other location can be provided by using an absolute template path).

The template script can contain any piece of code that can be executed by the underlying system. For example:

```
#!/bin/bash
echo "process started at `date`"
echo $STR
:
echo "process completed"
```

Tip: Note that the dollar character (\$) is interpreted as a Nextflow variable placeholder, when the script is run as a Nextflow template, while it is evaluated as a BASH variable when it is run alone. This can be very useful to test your script autonomously, i.e. independently from Nextflow execution. You only need to provide a BASH environment variable for each the Nextflow variable existing in your script. For example, it would be possible to execute the above script entering the following command in the shell terminal: `STR='foo' bash templates/my_script.sh`

4.1.4 Shell

Warning: This is an incubating feature. It may change in future Nextflow releases.

The `shell` block is a string statement that defines the *shell* command executed by the process to carry out its task. It is an alternative to the *Script* definition with an important difference, it uses the exclamation mark ! character as the variable placeholder for Nextflow variables in place of the usual dollar character.

In this way it is possible to use both Nextflow and BASH variables in the same piece of code without having to escape the latter and making process scripts more readable and easy to maintain. For example:

```
process myTask {
    input:
    val str from 'Hello', 'Hola', 'Bonjour'

    shell:
    '''
    echo User $USER says !{str}
    '''
}
```

In the above trivial example the `$USER` variable is managed by the BASH interpreter, while `!{str}` is handled as a process input variable managed by Nextflow.

Note:

- Shell script definition requires the use of single-quote ' delimited strings. When using double-quote " delimited strings, dollar variables are interpreted as Nextflow variables as usual. See [String interpolation](#).
 - Exclamation mark prefixed variables always need to be enclosed in curly brackets i.e. `!{str}` is a valid variable while `!str` is ignored.
 - Shell script supports the use of the file [Template](#) mechanism. The same rules are applied to the variables defined in the script template.
-

4.1.5 Native execution

Nextflow processes can execute native code other than system scripts as shown in the previous paragraphs.

This means that instead of specifying the process command to be executed as a string script, you can define it by providing one or more language statements, as you would do in the rest of the pipeline script. Simply starting the script definition block with the `exec:` keyword, for example:

```
x = Channel.from( 'a', 'b', 'c')

process simpleSum {
    input:
    val x

    exec:
    println "Hello Mr. $x"
}
```

Will display:

```
Hello Mr. b
Hello Mr. a
Hello Mr. c
```

4.2 Inputs

Nextflow processes are isolated from each other but can communicate between themselves sending values through channels.

The *input* block defines which channels the process is expecting to receive inputs data from. You can only define one input block at a time and it must contain one or more inputs declarations.

The input block follows the syntax shown below:

```
input:
  <input qualifier> <input name> [from <source channel>] [attributes]
```

An input definition starts with an input *qualifier* and the input *name*, followed by the keyword `from` and the actual channel over which inputs are received. Finally some input optional attributes can be specified.

Note: When the input name is the same as the channel name, the `from` part of the declaration can be omitted.

The input qualifier declares the *type* of data to be received. This information is used by Nextflow to apply the semantic rules associated to each qualifier and handle it properly depending on the target execution platform (grid, cloud, etc).

The qualifiers available are the ones listed in the following table:

| Qualifier | Semantic |
|-----------|---|
| val | Lets you access the received input value by its name in the process script. |
| env | Lets you use the received value to set an environment variable named as the specified input name. |
| file | Lets you handle the received value as a file, staging it properly in the execution context. |
| stdin | Lets you forward the received value to the process <i>stdin</i> special file. |
| set | Lets you handle a group of input values having one of the above qualifiers. |
| each | Lets you execute the process for each entry in the input collection. |

4.2.1 Input of generic values

The `val` qualifier allows you to receive data of any type as input. It can be accessed in the process script by using the specified input name, as shown in the following example:

```
num = Channel.from( 1, 2, 3 )

process basicExample {
    input:
    val x from num

    "echo process job $x"
}
```

In the above example the process is executed three times, each time a value is received from the channel `num` and used to process the script. Thus, it results in an output similar to the one shown below:

```
process job 3
process job 1
process job 2
```

Note: The *channel* guarantees that items are delivered in the same order as they have been sent - but - since the process is executed in a parallel manner, there is no guarantee that they are processed in the same order as they are received. In fact, in the above example, value 3 is processed before the others.

When the `val` has the same name as the channel from where the data is received, the `from` part can be omitted. Thus the above example can be written as shown below:

```
num = Channel.from( 1, 2, 3 )

process basicExample {
    input:
    val num

    "echo process job $num"
}
```

4.2.2 Input of files

The `file` qualifier allows you to receive a value as a file in the process execution context. This means that Nextflow will stage it in the process execution directory, and you can access it in the script by using the name specified in the input declaration. For example:

```
proteins = Channel.fromPath( '/some/path/*.fa' )

process blastThemAll {
  input:
  file query_file from proteins

  "blastp -query ${query_file} -db nr"
}
```

In the above example all the files ending with the suffix `.fa` are sent over the channel `proteins`. Then, these files are received by the process which will execute a *BLAST* query on each of them.

When the file input name is the same as the channel name, the `from` part of the input declaration can be omitted. Thus, the above example could be written as shown below:

```
proteins = Channel.fromPath( '/some/path/*.fa' )

process blastThemAll {
  input:
  file proteins

  "blastp -query $proteins -db nr"
}
```

It's worth noting that in the above examples, the name of the file in the file-system is not touched, you can access the file even without knowing its name because you can reference it in the process script by using the variable whose name is specified in the input file parameter declaration.

There may be cases where your task needs to use a file whose name is fixed, it does not have to change along with the actual provided file. In this case you can specify its name by specifying the `name` attribute in the input file parameter declaration, as shown in the following example:

```
input:
  file query_file name 'query.fa' from proteins
```

Or alternatively using a shorter syntax:

```
input:
  file 'query.fa' from proteins
```

Using this, the previous example can be re-written as shown below:

```
proteins = Channel.fromPath( '/some/path/*.fa' )

process blastThemAll {
  input:
  file 'query.fa' from proteins

  "blastp -query query.fa -db nr"
```

(continues on next page)

(continued from previous page)

}

What happens in this example is that each file, that the process receives, is staged with the name `query.fa` in a different execution context (i.e. the folder where the job is executed) and an independent process execution is launched.

Tip: This allows you to execute the process command various time without worrying the files names changing. In other words, *Nextflow* helps you write pipeline tasks that are self-contained and decoupled by the execution environment. This is also the reason why you should avoid whenever possible to use absolute or relative paths referencing files in your pipeline processes.

4.2.3 Multiple input files

A process can declare as input file a channel that emits a collection of values, instead of a simple value.

In this case, the script variable defined by the input file parameter will hold a list of files. You can use it as shown before, referring to all the files in the list, or by accessing a specific entry using the usual square brackets notation.

When a target file name is defined in the input parameter and a collection of files is received by the process, the file name will be appended by a numerical suffix representing its ordinal position in the list. For example:

```
fasta = Channel.fromPath( "/some/path/*.fa" ).buffer(size:3)

process blastThemAll {
    input:
        file 'seq' from fasta

    "echo seq*"
}
```

Will output:

```
seq1 seq2 seq3
seq1 seq2 seq3
...
```

The target input file name can contain the `*` and `?` wildcards, that can be used to control the name of staged files. The following table shows how the wildcards are replaced depending on the cardinality of the received input collection.

| Cardinality | Name pattern | Staged file names |
|-------------|--------------|---|
| any | * | named as the source file |
| 1 | file*.ext | file.ext |
| 1 | file?.ext | file1.ext |
| 1 | file??.ext | file01.ext |
| many | file*.ext | file1.ext, file2.ext, file3.ext, .. |
| many | file?.ext | file1.ext, file2.ext, file3.ext, .. |
| many | file??.ext | file01.ext, file02.ext, file03.ext, .. |
| many | dir/* | named as the source file, created in <code>dir</code> subdirectory |
| many | dir??/* | named as the source file, created in a progressively indexed subdirectory e.g. <code>dir01/</code> , <code>dir02/</code> , etc. |
| many | dir*/* | (as above) |

The following fragment shows how a wildcard can be used in the input file declaration:

```
fasta = Channel.fromPath( "/some/path/*.fa" ).buffer(size:3)

process blastThemAll {
    input:
        file 'seq?.fa' from fasta

    "cat seq1.fa seq2.fa seq3.fa"
}
```

Note: Rewriting input file names according to a named pattern is an extra feature and not at all obligatory. The normal file input constructs introduced in the *Input of files* section are valid for collections of multiple files as well. To handle multiple input files preserving the original file names, use the `*` wildcard as name pattern or a variable identifier.

4.2.4 Dynamic input file names

When the input file name is specified by using the `name` file clause or the short *string* notation, you are allowed to use other input values as variables in the file name string. For example:

```
process simpleCount {
    input:
        val x from species
        file "${x}.fa" from genomes

    """
    cat ${x}.fa | grep '>'
    """
}
```

In the above example, the input file name is set by using the current value of the `x` input value.

This allows the input files to be staged in the script working directory with a name that is coherent with the current execution context.

Tip: In most cases, you won't need to use dynamic file names, because each process is executed in its own private temporary directory, and input files are automatically staged to this directory by Nextflow. This guarantees that input files with the same name won't overwrite each other.

4.2.5 Input of type 'stdin'

The `stdin` input qualifier allows you the forwarding of the value received from a channel to the [standard input](#) of the command executed by the process. For example:

```
str = Channel.from('hello', 'hola', 'bonjour', 'ciao').map { it+'\n' }

process printAll {
    input:
    stdin str

    """
    cat -
    """
}
```

It will output:

```
hola
bonjour
ciao
hello
```

4.2.6 Input of type 'env'

The `env` qualifier allows you to define an environment variable in the process execution context based on the value received from the channel. For example:

```
str = Channel.from('hello', 'hola', 'bonjour', 'ciao')

process printEnv {

    input:
    env HELLO from str

    '''
    echo $HELLO world!
    '''
}
```

```
hello world!
ciao world!
bonjour world!
hola world!
```

4.2.7 Input of type ‘set’

The `set` qualifier allows you to group multiple parameters in a single parameter definition. It can be useful when a process receives, in input, tuples of values that need to be handled separately. Each element in the tuple is associated to a corresponding element with the `set` definition. For example:

```
tuple = Channel.from( [1, 'alpha'], [2, 'beta'], [3, 'delta'] )

process setExample {
    input:
    set val(x), file('latin.txt') from tuple

    """
    echo Processing $x
    cat - latin.txt > copy
    """
}
```

In the above example the `set` parameter is used to define the value `x` and the file `latin.txt`, which will receive a value from the same channel.

In the `set` declaration items can be defined by using the following qualifiers: `val`, `env`, `file` and `stdin`.

A shorter notation can be used by applying the following substitution rules:

| long | short |
|-----------------------------|-----------------------|
| <code>val(x)</code> | <code>x</code> |
| <code>file(x)</code> | (not supported) |
| <code>file('name')</code> | <code>'name'</code> |
| <code>file(x:'name')</code> | <code>x:'name'</code> |
| <code>stdin</code> | <code>'_'</code> |
| <code>env(x)</code> | (not supported) |

Thus the previous example could be rewritten as follows:

```
tuple = Channel.from( [1, 'alpha'], [2, 'beta'], [3, 'delta'] )

process setExample {
    input:
    set x, 'latin.txt' from tuple

    """
    echo Processing $x
    cat - latin.txt > copy
    """
}
```

File names can be defined in *dynamic* manner as explained in the [Dynamic input file names](#) section.

4.2.8 Input repeaters

The `each` qualifier allows you to repeat the execution of a process for each item in a collection, every time a new data is received. For example:

```

sequences = Channel.fromPath('*.fa')
methods = ['regular', 'expresso', 'psicoffee']

process alignSequences {
    input:
    file seq from sequences
    each mode from methods

    """
    t_coffee -in $seq -mode $mode > result
    """
}

```

In the above example every time a file of sequences is received as input by the process, it executes *three* tasks running a T-coffee alignment with a different value for the `mode` parameter. This is useful when you need to *repeat* the same task for a given set of parameters.

Since version 0.25+ input repeaters can be applied to files as well. For example:

```

sequences = Channel.fromPath('*.fa')
methods = ['regular', 'expresso']
libraries = [ file('PQ001.lib'), file('PQ002.lib'), file('PQ003.lib') ]

process alignSequences {
    input:
    file seq from sequences
    each mode from methods
    each file(lib) from libraries

    """
    t_coffee -in $seq -mode $mode -lib $lib > result
    """
}

```

Note: When multiple repeaters are declared, the process is executed for each *combination* of them.

In the latter example for any sequence input file emitted by the `sequences` channel are executed 6 alignments, 3 using the `regular` method against each library files, and other 3 by using the `expresso` method always against the same library files.

Hint: If you need to repeat the execution of a process over n-tuple of elements instead a simple values or files, create a channel combining the input values as needed to trigger the process execution multiple times. In this regard, see the *combine*, *cross* and *phase* operators.

4.3 Outputs

The *output* declaration block allows to define the channels used by the process to send out the results produced.

It can be defined at most one output block and it can contain one or more outputs declarations. The output block follows the syntax shown below:

```
output:
  <output qualifier> <output name> [into <target channel>[,channel,...]] [attribute [, .
  ↪.]]
```

Output definitions start by an output *qualifier* and the output *name*, followed by the keyword `into` and one or more channels over which outputs are sent. Finally some optional attributes can be specified.

Note: When the output name is the same as the channel name, the `into` part of the declaration can be omitted.

The qualifiers that can be used in the output declaration block are the ones listed in the following table:

| Qualifier | Semantic |
|---------------------|---|
| <code>val</code> | Sends variable's with the name specified over the output channel. |
| <code>file</code> | Sends a file produced by the process with the name specified over the output channel. |
| <code>stdout</code> | Sends the executed process <i>stdout</i> over the output channel. |
| <code>set</code> | Lets to send multiple values over the same output channel. |

4.3.1 Output values

The `val` qualifier allows to output a *value* defined in the script context. In a common usage scenario, this is a value which has been defined in the *input* declaration block, as shown in the following example:

```
methods = ['prot','dna', 'rna']

process foo {
  input:
    val x from methods

  output:
    val x into receiver

    """
    echo $x > file
    """
}

receiver.println { "Received: $it" }
```

Valid output values are value literals, input values identifiers, variables accessible in the process scope and value expressions. For example:

```
process foo {
  input:
    file fasta from 'dummy'

  output:
    val x into var_channel
    val 'BB11' into str_channel
    val "${fasta.baseName}.out" into exp_channel

  script:
    x = fasta.name
```

(continues on next page)

(continued from previous page)

```

"""
cat $x > file
"""
}

```

4.3.2 Output files

The `file` qualifier allows to output one or more files, produced by the process, over the specified channel. For example:

```

process randomNum {

    output:
    file 'result.txt' into numbers

    '''
    echo $RANDOM > result.txt
    '''

}

numbers.subscribe { println "Received: " + it.text }

```

In the above example the process, when executed, creates a file named `result.txt` containing a random number. Since a file parameter using the same name is declared between the outputs, when the task is completed that file is sent over the `numbers` channel. A downstream *process* declaring the same channel as *input* will be able to receive it.

Note: If the channel specified as output has not been previously declared in the pipeline script, it will implicitly be created by the output declaration itself.

4.3.3 Multiple output files

When an output file name contains a `*` or `?` wildcard character it is interpreted as a [glob](#) path matcher. This allows to *capture* multiple files into a list object and output them as a sole emission. For example:

```

process splitLetters {

    output:
    file 'chunk_*' into letters

    '''
    printf 'Hola' | split -b 1 - chunk_
    '''

}

letters
    .flatMap()
    .subscribe { println "File: ${it.name} => ${it.text}" }

```

It prints:

```
File: chunk_aa => H
File: chunk_ab => o
File: chunk_ac => l
File: chunk_ad => a
```

Note: In the above example the operator *flatMap* is used to transform the list of files emitted by the `letters` channel into a channel that emits each file object independently.

Some caveats on glob pattern behavior:

- Input files are not included in the list of possible matches.
- Glob pattern matches against both files and directories path.
- When a two stars pattern `**` is used to recurse across directories, only file paths are matched i.e. directories are not included in the result list.

Tip: By default all the files matching the specified glob pattern are emitted by the channel as a sole (list) item. It is also possible to emit each file as a sole item by adding the `mode flatten` attribute in the output file declaration.

By using the *mode* attribute the previous example can be re-written as show below:

```
process splitLetters {

    output:
    file 'chunk_*' into letters mode flatten

    '''
    printf 'Hola' | split -b 1 - chunk_
    '''
}

letters .subscribe { println "File: ${it.name} => ${it.text}" }
```

Read more about glob syntax at the following link [What is a glob?](#)

4.3.4 Dynamic output file names

When an output file name needs to be expressed dynamically, it is possible to define it using a dynamic evaluated string which references values defined in the input declaration block or in the script global context. For example:

```
process align {
    input:
    val x from species
    file seq from sequences

    output:
    file "${x}.aln" into genomes

    """
    t_coffee -in $seq > ${x}.aln
    """
}
```


In the above example, each time the process is executed an alignment file is produced whose name depends on the actual value of the `x` input.

Tip: The management of output files is a very common misunderstanding when using Nextflow. With other tools, it is generally necessary to organize the outputs files into some kind of directory structure or to guarantee a unique file name scheme, so that result files won't overwrite each other and that they can be referenced univocally by downstream tasks.

With Nextflow, in most cases, you don't need to take care of naming output files, because each task is executed in its own unique temporary directory, so files produced by different tasks can never override each other. Also meta-data can be associated with outputs by using the *set output* qualifier, instead of including them in the output file name.

To sum up, the use of output files with static names over dynamic ones is preferable whenever possible, because it will result in a simpler and more portable code.

4.3.5 Output 'stdout' special file

The `stdout` qualifier allows to *capture* the *stdout* output of the executed process and send it over the channel specified in the output parameter declaration. For example:

```
process echoSomething {
    output:
        stdout channel

    "echo Hello world!"
}

channel.subscribe { print "I say.. $it" }
```

4.3.6 Output 'set' of values

The `set` qualifier allows to send multiple values into a single channel. This feature is useful when you need to *group together* the results of multiple executions of the same process, as shown in the following example:

```
query = Channel.fromPath '*.fa'
species = Channel.from 'human', 'cow', 'horse'

process blast {
    input:
        val species
        file query

    output:
        set val(species), file('result') into blastOuts

    "blast -db nr -query $query" > result
}
```

In the above example a *BLAST* task is executed for each pair of `species` and `query` that are received. When the task completes a new tuple containing the value for `species` and the file `result` is sent to the `blastOuts` channel.

A *set* declaration can contain any combination of the following qualifiers, previously described: `val`, `file` and `stdout`.

Tip: Variable identifiers are interpreted as *values* while strings literals are interpreted as *files* by default, thus the above output *set* can be rewritten using a short notation as shown below.

```
output:
  set species, 'result' into blastOuts
```

File names can be defined in a dynamic manner as explained in the [Dynamic output file names](#) section.

4.3.7 Optional Output

In most cases a process is expected to generate output that is added to the output channel. However, there are situations where it is valid for a process to *not* generate output. In these cases `optional true` may be added to the output declaration, which tells Nextflow not to fail the process if the declared output is not created.

```
output:
  file("output.txt") optional true into outChannel
```

In this example, the process is normally expected to generate an `output.txt` file, but in the cases where the file is legitimately missing, the process does not fail. `outChannel` is only populated by those processes that do generate `output.txt`.

4.4 When

The `when` declaration allows you to define a condition that must be verified in order to execute the process. This can be any expression that evaluates a boolean value.

It is useful to enable/disable the process execution depending the state of various inputs and parameters. For example:

```
process find {
  input:
    file proteins
    val type from dbtype

  when:
    proteins.name =~ /^BB11.* / && type == 'nr'

  script:
    """
    blastp -query $proteins -db nr
    """
}
```

4.5 Directives

Using the *directive* declarations block you can provide optional settings that will affect the execution of the current process.

They must be entered at the top of the process *body*, before any other declaration blocks (i.e. `input`, `output`, etc) and have the following syntax:

```
name value [, value2 [...]]
```

Some directives are generally available to all processes, some others depends on the *executor* currently defined.

The directives are:

- *afterScript*
- *beforeScript*
- *cache*
- *cpus*
- *conda*
- *container*
- *containerOptions*
- *clusterOptions*
- *disk*
- *echo*
- *errorStrategy*
- *executor*
- *ext*
- *queue*
- *label*
- *maxErrors*
- *maxForks*
- *maxRetries*
- *memory*
- *module*
- *penv*
- *pod*
- *publishDir*
- *scratch*
- *stageInMode*
- *stageOutMode*
- *storeDir*
- *tag*
- *time*
- *validExitStatus*

4.5.1 afterScript

The `afterScript` directive allows you to execute a custom (BASH) snippet immediately *after* the main process has run. This may be useful to clean up your staging area.

4.5.2 beforeScript

The `beforeScript` directive allows you to execute a custom (BASH) snippet *before* the main process script is run. This may be useful to initialise the underlying cluster environment or for other custom initialisation.

For example:

```
process foo {  
  
    beforeScript 'source /cluster/bin/setup'  
  
    """  
    echo bar  
    """  
  
}
```

4.5.3 cache

The `cache` directive allows you to store the process results to a local cache. When the cache is enabled *and* the pipeline is launched with the *resume* option, any following attempt to execute the process, along with the same inputs, will cause the process execution to be skipped, producing the stored data as the actual results.

The caching feature generates a unique *key* by indexing the process script and inputs. This key is used identify univocally the outputs produced by the process execution.

The cache is enabled by default, you can disable it for a specific process by setting the `cache` directive to `false`. For example:

```
process noCacheThis {  
    cache false  
  
    script:  
    <your command string here>  
}
```

The `cache` directive possible values are shown in the following table:

| Value | Description |
|-------------------|---|
| false | Disable cache feature. |
| true (default) | Cache process outputs. Input files are indexed by using the meta-data information (name, size and last update timestamp). |
| 'deep' | Cache process outputs. Input files are indexed by their content. |

4.5.4 conda

The `conda` directive allows for the definition of the process dependencies using the [Conda](#) package manager.

Nextflow automatically sets up an environment for the given package names listed by in the `conda` directive. For example:

```
process foo {
  conda 'bwa=0.7.15'

  '''
  your_command --here
  '''
}
```

Multiple packages can be specified separating them with a blank space eg. `bwa=0.7.15 fastqc=0.11.5`. The name of the channel from where a specific package needs to be downloaded can be specified using the usual Conda notation i.e. prefixing the package with the channel name as shown here `bioconda::bwa=0.7.15`.

The `conda` directory also allows the specification of a Conda environment file path or the path of an existing environment directory. See the [Conda environments](#) page for further details.

4.5.5 container

The `container` directive allows you to execute the process script in a [Docker](#) container.

It requires the Docker daemon to be running in machine where the pipeline is executed, i.e. the local machine when using the *local* executor or the cluster nodes when the pipeline is deployed through a *grid* executor.

For example:

```
process runThisInDocker {

  container 'dockerbox:tag'

  """
  <your holy script here>
  """

}
```

Simply replace in the above script `dockerbox:tag` with the Docker image name you want to use.

Tip: This can be very useful to execute your scripts into a replicable self-contained environment or to deploy your pipeline in the cloud.

Note: This directive is ignore for processes *executed natively*.

4.5.6 containerOptions

The `containerOptions` directive allows you to specify any container execution option supported by the underlying container engine (ie. Docker, Singularity, etc). This can be useful to provide container settings only for a specific process e.g. mount a custom path:

```
process runThisWithDocker {  
  
    container 'busybox:latest'  
    containerOptions '--volume /data/db:/db'  
  
    output: file 'output.txt'  
  
    '''  
    your_command --data /db > output.txt  
    '''  
}
```

Warning: This feature is not supported by *AWS Batch* and *Kubernetes* executors.

4.5.7 cpus

The `cpus` directive allows you to define the number of (logical) CPU required by the process' task. For example:

```
process big_job {  
  
    cpus 8  
    executor 'sge'  
  
    '''  
    blastp -query input_sequence -num_threads ${task.cpus}  
    '''  
}
```

This directive is required for tasks that execute multi-process or multi-threaded commands/tools and it is meant to reserve enough CPUs when a pipeline task is executed through a cluster resource manager.

See also: *penv*, *memory*, *time*, *queue*, *maxForks*

4.5.8 clusterOptions

The `clusterOptions` directive allows the usage of any *native* configuration option accepted by your cluster submit command. You can use it to request non-standard resources or use settings that are specific to your cluster and not supported out of the box by Nextflow.

Note: This directive is taken in account only when using a grid based executor: *SGE*, *LSF*, *SLURM*, *PBS/Torque* and *HTCondor* executors.

4.5.9 disk

The `disk` directive allows you to define how much local disk storage the process is allowed to use. For example:

```
process big_job {  
  
    disk '2 GB'
```

(continues on next page)

(continued from previous page)

```

    executor 'cirrus'

    """
    your task script here
    """
}

```

The following memory unit suffix can be used when specifying the disk value:

| Unit | Description |
|------|-------------|
| B | Bytes |
| KB | Kilobytes |
| MB | Megabytes |
| GB | Gigabytes |
| TB | Terabytes |

Note: This directive currently is taken in account only by the *Ignite* and the *HTCondor* executors.

See also: *cpus*, *memory time*, *queue* and *Dynamic computing resources*.

4.5.10 echo

By default the *stdout* produced by the commands executed in all processes is ignored. Setting the `echo` directive to `true` you can forward the process *stdout* to the current top running process *stdout* file, showing it in the shell terminal.

For example:

```

process sayHello {
    echo true

    script:
    "echo Hello"
}

```

```
Hello
```

Without specifying `echo true` you won't see the `Hello` string printed out when executing the above example.

4.5.11 errorStrategy

The `errorStrategy` directive allows you to define how an error condition is managed by the process. By default when an error status is returned by the executed script, the process stops immediately. This in turn forces the entire pipeline to terminate.

Table of available error strategies:

| Name | Executor |
|------------------------|--|
| <code>terminate</code> | Terminates the execution as soon as an error condition is reported. Pending jobs are killed (default) |
| <code>finish</code> | Initiates an orderly pipeline shutdown when an error condition is raised, waiting the completion of any submitted job. |
| <code>ignore</code> | Ignores processes execution errors. |
| <code>retry</code> | Re-submit for execution a process returning an error condition. |

When setting the `errorStrategy` directive to `ignore` the process doesn't stop on an error condition, it just reports a message notifying you of the error event.

For example:

```
process ignoreAnyError {
    errorStrategy 'ignore'

    script:
    <your command string here>
}
```

Tip: By definition a command script fails when it ends with a non-zero exit status. To change this behavior see [validExitStatus](#).

The `retry` *error strategy*, allows you to re-submit for execution a process returning an error condition. For example:

```
process retryIfFail {
    errorStrategy 'retry'

    script:
    <your command string here>
}
```

The number of times a failing process is re-executed is defined by the `maxRetries` and `maxErrors` directives.

4.5.12 executor

The *executor* defines the underlying system where processes are executed. By default a process uses the executor defined globally in the `nextflow.config` file.

The `executor` directive allows you to configure what executor has to be used by the process, overriding the default configuration. The following values can be used:

| Name | Executor |
|---------------------|--|
| <code>local</code> | The process is executed in the computer where <i>Nextflow</i> is launched. |
| <code>sge</code> | The process is executed using the Sun Grid Engine / Open Grid Engine . |
| <code>uge</code> | The process is executed using the Univa Grid Engine job scheduler. |
| <code>lsf</code> | The process is executed using the Platform LSF job scheduler. |
| <code>slurm</code> | The process is executed using the SLURM job scheduler. |
| <code>pbs</code> | The process is executed using the PBS/Torque job scheduler. |
| <code>condor</code> | The process is executed using the HTCondor job scheduler. |
| <code>nqsii</code> | The process is executed using the NQSII job scheduler. |
| <code>ignite</code> | The process is executed using the Apache Ignite cluster. |
| <code>k8s</code> | The process is executed using the Kubernetes cluster. |

The following example shows how to set the process's executor:

```
process doSomething {
    executor 'sge'

    script:
    <your script here>
}
```

Note: Each executor provides its own set of configuration options that can be set in the *directive* declarations block. See [Executors](#) section to read about specific executor directives.

4.5.13 ext

The `ext` is a special directive used as *namespace* for user custom process directives. This can be useful for advanced configuration options. For example:

```
process mapping {
    container "biocontainers/star:${task.ext.version}"

    input:
    file genome from genome_file
    set sampleId, file(reads) from reads_ch

    """
    STAR --genomeDir $genome --readFilesIn $reads
    """
}
```

In the above example, the process uses a container whose version is controlled by the `ext.version` property. This can be defined in the `nextflow.config` file as shown below:

```
process.ext.version = '2.5.3'
```

4.5.14 maxErrors

The `maxErrors` directive allows you to specify the maximum number of times a process can fail when using the *retry error strategy*. By default this directive is disabled, you can set it as shown in the example below:

```
process retryIfFail {
    errorStrategy 'retry'
    maxErrors 5

    """
    echo 'do this as that .. '
    """
}
```

Note: This setting considers the **total** errors accumulated for a given process, across all instances. If you want to

control the number of times a process **instance** (aka task) can fail, use `maxRetries`.

See also: *errorStrategy* and *maxRetries*.

4.5.15 maxForks

The `maxForks` directive allows you to define the maximum number of process instances that can be executed in parallel. By default this value is equals to the number of CPU cores available minus 1.

If you want to execute a process in a sequential manner, set this directive to one. For example:

```
process doNotParallelizeIt {  
  
    maxForks 1  
  
    '''  
    <your script here>  
    '''  
  
}
```

4.5.16 maxRetries

The `maxRetries` directive allows you to define the maximum number of times a process instance can be re-submitted in case of failure. This value is applied only when using the `retry error strategy`. By default only one retry is allowed, you can increase this value as shown below:

```
process retryIfFail {  
    errorStrategy 'retry'  
    maxRetries 3  
  
    """  
    echo 'do this as that .. '  
    """  
  
}
```

Note: There is a subtle but important difference between `maxRetries` and the `maxErrors` directive. The latter defines the total number of errors that are allowed during the process execution (the same process can launch different execution instances), while the `maxRetries` defines the maximum number of times the same process execution can be retried in case of an error.

See also: *errorStrategy* and *maxErrors*.

4.5.17 memory

The `memory` directive allows you to define how much memory the process is allowed to use. For example:

```
process big_job {  
  
    memory '2 GB'  
    executor 'sge'
```

(continues on next page)

(continued from previous page)

```

    """
    your task script here
    """
}

```

The following memory unit suffix can be used when specifying the memory value:

| Unit | Description |
|------|-------------|
| B | Bytes |
| KB | Kilobytes |
| MB | Megabytes |
| GB | Gigabytes |
| TB | Terabytes |

See also: *cpus*, *time*, *queue* and *Dynamic computing resources*.

4.5.18 module

Environment Modules is a package manager that allows you to dynamically configure your execution environment and easily switch between multiple versions of the same software tool.

If it is available in your system you can use it with Nextflow in order to configure the processes execution environment in your pipeline.

In a process definition you can use the `module` directive to load a specific module version to be used in the process execution environment. For example:

```

process basicExample {

    module 'ncbi-blast/2.2.27'

    """
    blastp -query <etc..>
    """
}

```

You can repeat the `module` directive for each module you need to load. Alternatively multiple modules can be specified in a single `module` directive by separating all the module names by using a `:` (colon) character as shown below:

```

process manyModules {

    module 'ncbi-blast/2.2.27:t_coffee/10.0:clustalw/2.1'

    """
    blastp -query <etc..>
    """
}

```

4.5.19 penv

The `penv` directive allows you to define the *parallel environment* to be used when submitting a parallel task to the *SGE* resource manager. For example:

```
process big_job {  
  
    cpus 4  
    penv 'smp'  
    executor 'sge'  
  
    """  
    blastp -query input_sequence -num_threads ${task.cpus}  
    """  
}
```

This configuration depends on the parallel environment provided by your grid engine installation. Refer to your cluster documentation or contact your admin to learn more about this.

Note: This setting is available when using the *SGE* executor.

See also: *cpus*, *memory*, *time*

4.5.20 pod

The `pod` directive allows the definition of pods specific settings, such as environment variables, secrets and config maps when using the *Kubernetes* executor.

For example:

```
process your_task {  
    pod env: 'FOO', value: 'bar'  
  
    '''  
    echo $FOO  
    '''  
}
```

The above snippet defines an environment variable named `FOO` which value is `bar`.

The `pod` directive allows the definition of the following options:

| | |
|---|---|
| env: <E>, value: <V> | Defines an environment variable with name E and whose value is given by the V string. |
| env: <E>, config: <C/K> | Defines an environment variable with name E and whose value is given by the entry associated to the key with name K in the ConfigMap with name C. |
| env: <E>, secret: <S/K> | Defines an environment variable with name E and whose value is given by the entry associated to the key with name K in the Secret with name S. |
| config: <C/K>, mountPath: </absolute/path> | The content of the ConfigMap with name C with key K is made available to the path /absolute/path. When the key component is omitted the path is interpreted as a directory and all the <i>ConfigMap</i> entries are exposed in that path. |
| secret: <S/K>, mountPath: </absolute/path> | The content of the Secret with name S with key K is made available to the path /absolute/path. When the key component is omitted the path is interpreted as a directory and all the <i>Secret</i> entries are exposed in that path. |
| volumeClaim: <V>, mountPath: </absolute/path> | Mounts a Persistent volume claim with name V to the specified path location. |

When defined in the Nextflow configuration file, a pod setting can be defined using the canonical associative array syntax. For example:

```
process {
  pod = [env: 'FOO', value: 'bar']
}
```

When more than one setting needs to be provides they must be enclosed in a list definition as shown below:

```
process {
  pod = [ [env: 'FOO', value: 'bar'], [secret: 'my-secret/key1', mountPath: '/etc/
↪file.txt'] ]
}
```

4.5.21 publishDir

The `publishDir` directive allows you to publish the process output files to a specified folder. For example:

```
process foo {

  publishDir '/data/chunks'

  output:
  file 'chunk_*' into letters

  '''
  printf 'Hola' | split -b 1 - chunk_
  '''
}
```

The above example splits the string `Hola` into file chunks of a single byte. When complete the `chunk_*` output files are published into the `/data/chunks` folder.

Tip: The `publishDir` directive can be specified more than one time in to publish the output files to different target directories. This feature requires version 0.29.0 or higher.

By default files are published to the target folder creating a *symbolic link* for each process output that links the file produced into the process working directory. This behavior can be modified using the `mode` parameter.

Table of optional parameters that can be used with the `publishDir` directive:

| Name | Description |
|------------------------|---|
| <code>mode</code> | The file publishing method. See the following table for possible values. |
| <code>overwrite</code> | When <code>true</code> any existing file in the specified folder will be overridden (default: <code>true</code> during normal pipeline execution and <code>false</code> when pipeline execution is <i>resumed</i>). |
| <code>pattern</code> | Specifies a <code>glob</code> file pattern that selects which files to publish from the overall set of output files. |
| <code>path</code> | Specifies the directory where files need to be published. Note: the syntax <code>publishDir '/some/dir'</code> is a shortcut for <code>publishDir path: '/some/dir'</code> . |
| <code>saveAs</code> | A closure which, given the name of the file being published, returns the actual file name or a full path where the file is required to be stored. This can be used to rename or change the destination directory of the published files dynamically by using a custom strategy. Return the value <code>null</code> from the closure to <i>not</i> publish a file. This is useful when the process has multiple output files, but you want to publish only some of them. |

Table of publish modes:

| Mode | Description |
|----------------------------|--|
| <code>sym-link</code> | Creates a <i>symbolic link</i> in the published directory for each process output file (default). |
| <code>link</code> | Creates a <i>hard link</i> in the published directory for each process output file. |
| <code>copy</code> | Copies the output files into the published directory. |
| <code>copy-NoFollow</code> | Copies the output files into the published directory without following symlinks ie. copies the links themselves. |
| <code>move</code> | Moves the output files into the published directory. Note: this is only supposed to be used for a <i>terminating</i> process i.e. a process whose output is not consumed by any other downstream process. |

Note: The `mode` value needs to be specified as a string literal i.e. enclosed by quote characters. Multiple parameters need to be separated by a colon character. For example:

```
process foo {
    publishDir '/data/chunks', mode: 'copy', overwrite: false

    output:
    file 'chunk_*' into letters

    '''
    printf 'Hola' | split -b 1 - chunk_
    '''
}
```

Warning: Files are copied into the specified directory in an *asynchronous* manner, thus they may not be immediately available in the published directory at the end of the process execution. For this reason files published by a process must not be accessed by other downstream processes.

4.5.22 queue

The `queue` directive allows you to set the *queue* where jobs are scheduled when using a grid based executor in your pipeline. For example:

```
process grid_job {
    queue 'long'
    executor 'sge'

    """
    your task script here
    """
}
```

Multiple queues can be specified by separating their names with a comma for example:

```
process grid_job {
    queue 'short,long,cn-el6'
    executor 'sge'

    """
    your task script here
    """
}
```

Note: This directive is taken in account only by the following executors: *SGE*, *LSF*, *SLURM* and *PBS/Torque* executors.

4.5.23 label

The `label` directive allows the annotation of processes with mnemonic identifier of your choice. For example:

```
process bigTask {
    label 'big_mem'

    '''
    <task script>
    '''
}
```

The same label can be applied to more than a process and multiple labels can be applied to the same process using the `label` directive more than one time.

Note: A label must consist of alphanumeric characters or `_`, must start with an alphabetic character and must end with an alphanumeric character.

Labels are useful to organise workflow processes in separate groups which can be referenced in the configuration file to select and configure subset of processes having similar computing requirements.

See the *Process selectors* documentation for details.

4.5.24 scratch

The `scratch` directive allows you to execute the process in a temporary folder that is local to the execution node.

This is useful when your pipeline is launched by using a *grid* executor, because it permits to decrease the NFS overhead by running the pipeline processes in a temporary directory in the local disk of the actual execution node. Only the files declared as output in the process definition will be copied in the pipeline working area.

In its basic form simply specify `true` at the directive value, as shown below:

```
process simpleTask {  
  
    scratch true  
  
    output:  
    file 'data_out'  
  
    '''  
    <task script>  
    '''  
}
```

By doing this, it tries to execute the script in the directory defined by the variable `$TMPDIR` in the execution node. If this variable does not exist, it will create a new temporary directory by using the Linux command `mktemp`.

A custom environment variable, other than `$TMPDIR`, can be specified by simply using it as the `scratch` value, for example:

```
scratch '$MY_GRID_TMP'
```

Note, it must be wrapped by single quotation characters, otherwise the variable will be evaluated in the pipeline script context.

You can also provide a specific folder path as `scratch` value, for example:

```
scratch '/tmp/my/path'
```

By doing this, a new temporary directory will be created in the specified path each time a process is executed.

Finally, when the `ram-disk` string is provided as `scratch` value, the process will be execute in the node RAM virtual disk.

Summary of allowed values:

| scratch | Description |
|-------------------------|---|
| false | Do not use the scratch folder. |
| true | Creates a scratch folder in the directory defined by the <code>\$TMPDIR</code> variable; fallback to <code>mktemp /tmp</code> if that variable do not exists. |
| <code>\$YOUR_VAR</code> | Creates a scratch folder in the directory defined by the <code>\$YOUR_VAR</code> environment variable; fallback to <code>mktemp /tmp</code> if that variable do not exists. |
| <code>/my/tmp</code> | Creates a scratch folder in the specified directory. |
| ram-disk | Creates a scratch folder in the RAM disk <code>/dev/shm/</code> (experimental). |

4.5.25 storeDir

The `storeDir` directive allows you to define a directory that is used as *permanent* cache for your process results.

In more detail, it affects the process execution in two main ways:

1. The process is executed only if the files declared in the *output* clause do not exist in the directory specified by the `storeDir` directive. When the files exist the process execution is skipped and these files are used as the actual process result.
2. Whenever a process is successfully completed the files listed in the *output* declaration block are copied into the directory specified by the `storeDir` directive.

The following example shows how to use the `storeDir` directive to create a directory containing a BLAST database for each species specified by an input parameter:

```
genomes = Channel.fromPath(params.genomes)

process formatBlastDatabases {

    storeDir '/db/genomes'

    input:
    file species from genomes

    output:
    file "${dbName}.*" into blastDb

    script:
    dbName = species.baseName
    """
    makeblastdb -dbtype nucl -in ${species} -out ${dbName}
    """
}
```

Warning: The `storeDir` directive is meant for long term process caching and should not be used to output the files produced by a process to a specific folder or organise result data in *semantic* directory structure. In these cases you may use the `publishDir` directive instead.

Note: The use of AWS S3 path is supported however it requires the installation of the [AWS CLI tool](#) (ie. `aws`) in the target computing node.

4.5.26 stageInMode

The `stageInMode` directive defines how input files are staged-in to the process work directory. The following values are allowed:

| Value | Description |
|---------|---|
| copy | Input files are staged in the process work directory by creating a copy. |
| link | Input files are staged in the process work directory by creating an (hard) link for each of them. |
| symlink | Input files are staged in the process work directory by creating an symlink for each of them (default). |

4.5.27 stageOutMode

The `stageOutMode` directive defines how output files are staged-out from the scratch directory to the process work directory. The following values are allowed:

| Value | Description |
|-------|---|
| copy | Output files are copied from the scratch directory to the work directory. |
| move | Output files are moved from the scratch directory to the work directory. |
| rsync | Output files are copied from the scratch directory to the work directory by using the <code>rsync</code> utility. |

See also: *scratch*.

4.5.28 tag

The `tag` directive allows you to associate each process executions with a custom label, so that it will be easier to identify them in the log file or in the trace execution report. For example:

```
process foo {
  tag { code }

  input:
  val code from 'alpha', 'gamma', 'omega'

  """
  echo $code
  """
}
```

The above snippet will print a log similar to the following one, where process names contain the tag value:

```
[6e/28919b] Submitted process > foo (alpha)
[d2/1c6175] Submitted process > foo (gamma)
[1c/3ef220] Submitted process > foo (omega)
```

See also *Trace execution report*

4.5.29 time

The `time` directive allows you to define how long a process is allowed to run. For example:

```
process big_job {

  time '1h'

  """
  your task script here
  """
}
```

The following time unit suffix can be used when specifying the duration value:

| Unit | Description |
|------|-------------|
| s | Seconds |
| m | Minutes |
| h | Hours |
| d | Days |

Note: This directive is taken in account only when using one of the following grid based executors: *SGE*, *LSF*, *SLURM*, *PBS/Torque*, *HTCondor* and *AWS Batch* executors.

See also: *cpus*, *memory*, *queue* and *Dynamic computing resources*.

4.5.30 validExitStatus

A process is terminated when the executed command returns an error exit status. By default any error status other than 0 is interpreted as an error condition.

The `validExitStatus` directive allows you to fine control which error status will represent a successful command execution. You can specify a single value or multiple values as shown in the following example:

```
process returnOk {
    validExitStatus 0,1,2

    script:
    """
    echo Hello
    exit 1
    """
}
```

In the above example, although the command script ends with a 1 exit status, the process will not return an error condition because the value 1 is declared as a *valid* status in the `validExitStatus` directive.

4.5.31 Dynamic directives

A directive can be assigned *dynamically*, during the process execution, so that its actual value can be evaluated depending on the value of one, or more, process' input values.

In order to be defined in a dynamic manner the directive's value needs to be expressed by using a *closure* statement, as in the following example:

```
process foo {

    executor 'sge'
    queue { entries > 100 ? 'long' : 'short' }

    input:
    set entries, file(x) from data

    script:
    """
    < your job here >
    """
}
```

In the above example the *queue* directive is evaluated dynamically, depending on the input value `entries`. When it is bigger than 100, jobs will be submitted to the queue `long`, otherwise the `short` one will be used.

All directives can be assigned to a dynamic value except the following:

- *executor*

- *maxForks*

Note: You can retrieve the current value of a dynamic directive in the process script by using the implicit variable `task` which holds the directive values defined in the current process instance.

For example:

```
process foo {

    queue { entries > 100 ? 'long' : 'short' }

    input:
    set entries, file(x) from data

    script:
    """
    echo Current queue: ${task.queue}
    """
}
```

4.5.32 Dynamic computing resources

It's very common scenario that different instances of the same process may have a very different needs in terms of computing resources. In such situation requesting, for example, an amount of memory too low will cause some tasks to fail, instead using an higher limit that fits all the tasks execution could significantly decrease the execution priority of your jobs.

The *Dynamic directives* evaluation feature can be used to modify the amount of computing resources request in case of a process failure and try to re-execute it using an higher limit. For example:

```
process foo {

    memory { 2.GB * task.attempt }
    time { 1.hour * task.attempt }

    errorStrategy { task.exitStatus == 140 ? 'retry' : 'terminate' }
    maxRetries 3

    script:
    <your job here>

}
```

In the above example the *memory* and execution *time* limits are defined dynamically. The first time the process is executed the `task.attempt` is set to 1, thus it will request a two GB of memory and one hour of maximum execution time.

If the task execution fails reporting an exit status equals 140, the task is re-submitted (otherwise terminates immediately). This time the value of `task.attempt` is 2, thus increasing the amount of the memory to four GB and the time to 2 hours, and so on.

The directive *maxRetries* sets the maximum number of times the same task can be re-executed.

Nextflow is based on the Dataflow programming model in which processes communicate through channels.

A *channel* is a non-blocking unidirectional FIFO queue which connects two processes. It has two properties:

1. Sending a message is an *asynchronous* operation which completes immediately, without having to wait for the receiving process.
2. Receiving data is a blocking operation which stops the receiving process until the message has arrived.

5.1 Channel factory

Channels may be created implicitly by the process output(s) declaration or explicitly using the following channel factory methods.

The available factory methods are:

- *create*
- *empty*
- *from*
- *fromPath*
- *fromFilePairs*
- *value*
- *watchPath*

5.1.1 create

Creates a new *channel* by using the `create` method, as shown below:

```
channelObj = Channel.create()
```

5.1.2 from

The `from` method allows you to create a channel emitting any sequence of values that are specified as the method argument, for example:

```
ch = Channel.from( 1, 3, 5, 7 )
ch.subscribe { println "value: $it" }
```

The first line in this example creates a variable `ch` which holds a channel object. This channel emits the values specified as a parameter in the `from` method. Thus the second line will print the following:

```
value: 1
value: 3
value: 5
value: 7
```

The following example shows how to create a channel from a *range* of numbers or strings:

```
zeroToNine = Channel.from( 0..9 )
strings = Channel.from( 'A'..'Z' )
```

Note: Note that when the `from` argument is an object implementing the (Java) `Collection` interface, the resulting channel emits the collection entries as individual emissions.

Thus the following two declarations produce an identical result even though in the first case the items are specified as multiple arguments while in the second case as a single list object argument:

```
Channel.from( 1, 3, 5, 7, 9 )
Channel.from( [1, 3, 5, 7, 9] )
```

But when more than one argument is provided, they are always managed as *single* emissions. Thus, the following example creates a channel emitting three entries each of which is a list containing two elements:

```
Channel.from( [1, 2], [5, 6], [7, 9] )
```

5.1.3 value

This method creates a dataflow *variable* that is a channel to which one entry, at most, can be bound. An optional not null value can be specified as a parameter, which is bound to the newly created channel. For example:

```
expl1 = Channel.value()
expl2 = Channel.value( 'Hello there' )
expl3 = Channel.value( [1, 2, 3, 4, 5] )
```

The first line in the example creates an ‘empty’ variable. The second line creates a channel and binds a string to it. Finally the last one creates a channel and binds a list object to it that will be emitted as a sole emission.

5.1.4 fromPath

You can create a channel emitting one or more file paths by using the `fromPath` method and specifying a path string as an argument. For example:

```
myFileChannel = Channel.fromPath( '/data/some/bigfile.txt' )
```

The above line creates a channel and binds to it a `Path` item referring the specified file.

Note: It does not check the file existence.

Whenever the `fromPath` argument contains a `*` or `?` wildcard character it is interpreted as a `glob` path matcher. For example:

```
myFileChannel = Channel.fromPath( '/data/big/*.txt' )
```

This example creates a channel and emits as many `Path` items as there are files with `txt` extension in the `/data/big` folder.

Tip: Two asterisks, i.e. `**`, works like `*` but crosses directory boundaries. This syntax is generally used for matching complete paths. Curly brackets specify a collection of sub-patterns.

For example:

```
files = Channel.fromPath( 'data/**/*.fa' )
moreFiles = Channel.fromPath( 'data/**/*.*fa' )
pairFiles = Channel.fromPath( 'data/file_{1,2}.fq' )
```

The first line returns a channel emitting the files ending with the suffix `.fa` in the `data` folder *and* recursively in all its sub-folders. While the second one only emits the files which have the same suffix in *any* sub-folder in the `data` path. Finally the last example emits two files: `data/file_1.fq` and `data/file_2.fq`.

Note: As in Linux BASH the `*` wildcard does not match against hidden files (i.e. files whose name start with a `.` character).

In order to include hidden files, you need to start your pattern with a period character or specify the `hidden: true` option. For example:

```
expl1 = Channel.fromPath( '/path/.*' )
expl2 = Channel.fromPath( '/path/*.*fa' )
expl3 = Channel.fromPath( '/path/*', hidden: true )
```

The first example returns all hidden files in the specified path. The second one returns all hidden files ending with the `.fa` suffix. Finally the last example returns all files (hidden and non-hidden) in that path.

By default a `glob` pattern only looks for *regular file* paths that match the specified criteria, i.e. it won't return directory paths.

You may use the parameter `type` specifying the value `file`, `dir` or `any` in order to define what kind of paths you want. For example:

```
myFileChannel = Channel.fromPath( '/path/*b', type: 'dir' )
myFileChannel = Channel.fromPath( '/path/a*', type: 'any' )
```

The first example will return all *directory* paths ending with the `b` suffix, while the second will return any file and directory starting with a `a` prefix.

| Name | Description |
|---------------------------|--|
| <code>glob</code> | When <code>true</code> interprets characters <code>*</code> , <code>?</code> , <code>[]</code> and <code>{ }</code> as glob wildcards, otherwise handles them as normal characters (default: <code>true</code>) |
| <code>type</code> | Type of paths returned, either <code>file</code> , <code>dir</code> or <code>any</code> (default: <code>file</code>) |
| <code>hidden</code> | When <code>true</code> includes hidden files in the resulting paths (default: <code>false</code>) |
| <code>maxDepth</code> | Maximum number of directory levels to visit (default: <i>no limit</i>) |
| <code>fol-lowLinks</code> | When <code>true</code> it follows symbolic links during directories tree traversal, otherwise they are managed as files (default: <code>true</code>) |
| <code>relative</code> | When <code>true</code> returned paths are relative to the top-most common directory (default: <code>false</code>) |

5.1.5 fromFilePairs

The `fromFilePairs` method creates a channel emitting the file pairs matching a `glob` pattern provided by the user. The matching files are emitted as tuples in which the first element is the grouping key of the matching pair and the second element is the list of files (sorted in lexicographical order). For example:

```
Channel
  .fromFilePairs('/my/data/SRR*_{1,2}.fastq')
  .println()
```

It will produce an output similar to the following:

```
[SRR493366, [/my/data/SRR493366_1.fastq, /my/data/SRR493366_2.fastq]]
[SRR493367, [/my/data/SRR493367_1.fastq, /my/data/SRR493367_2.fastq]]
[SRR493368, [/my/data/SRR493368_1.fastq, /my/data/SRR493368_2.fastq]]
[SRR493369, [/my/data/SRR493369_1.fastq, /my/data/SRR493369_2.fastq]]
[SRR493370, [/my/data/SRR493370_1.fastq, /my/data/SRR493370_2.fastq]]
[SRR493371, [/my/data/SRR493371_1.fastq, /my/data/SRR493371_2.fastq]]
```

Note: The glob pattern must contain at least a star wildcard character.

Alternatively it is possible to implement a custom file pair grouping strategy providing a closure which, given the current file as parameter, returns the grouping key. For example:

```
Channel
  .fromFilePairs('/some/data/*', size: -1) { file -> file.extension }
  .println { ext, files -> "Files with the extension $ext are $files" }
```

Table of optional parameters available:

| Name | Description |
|---------------------------|---|
| <code>type</code> | Type of paths returned, either <code>file</code> , <code>dir</code> or <code>any</code> (default: <code>file</code>) |
| <code>hidden</code> | When <code>true</code> includes hidden files in the resulting paths (default: <code>false</code>) |
| <code>maxDepth</code> | Maximum number of directory levels to visit (default: <i>no limit</i>) |
| <code>fol-lowLinks</code> | When <code>true</code> it follows symbolic links during directories tree traversal, otherwise they are managed as files (default: <code>true</code>) |
| <code>size</code> | Defines the number of files each emitted item is expected to hold (default: 2). Set to <code>-1</code> for any. |
| <code>flat</code> | When <code>true</code> the matching files are produced as sole elements in the emitted tuples (default: <code>false</code>). |

5.1.6 watchPath

The `watchPath` method watches a folder for one or more files matching a specified pattern. As soon as there is a file that meets the specified condition, it is emitted over the channel that is returned by the `watchPath` method. The condition on files to watch can be specified by using `*` or `?` wildcard characters i.e. by specifying a [glob](#) path matching criteria.

For example:

```
Channel
  .watchPath( '/path/*.fa' )
  .subscribe { println "Fasta file: $it" }
```

By default it watches only for new files created in the specified folder. Optionally, it is possible to provide a second argument that specifies what event(s) to watch. The supported events are:

| Name | Description |
|--------|---------------------------------|
| create | A new file is created (default) |
| modify | A file is modified |
| delete | A file is deleted |

You can specify more than one of these events by using a comma separated string as shown below:

```
Channel
  .watchPath( '/path/*.fa', 'create,modify' )
  .subscribe { println "File created or modified: $it" }
```

Warning: The `watchPath` factory waits endlessly for files that match the specified pattern and event(s). Thus, whenever you use it in your script, the resulting pipeline will never finish.

See also: [fromPath](#) factory method.

5.1.7 empty

The `empty` factory method, by definition, creates a channel that doesn't emit any value.

See also: [ifEmpty](#) and [close](#) operators.

5.2 Binding values

Since in *Nextflow* channels are implemented using *dataflow* variables or queues. Thus sending a message is equivalent to *bind* a value to object representing the communication channel.

5.2.1 bind()

Channel objects provide a `bind()` method which is the basic operation to send a message over the channel. For example:

```
myChannel = Channel.create()
myChannel.bind( 'Hello world' )
```

5.2.2 operator <<

The operator << is just a syntax sugar for the `bind()` method. Thus, the following example produce an identical result as the previous one:

```
myChannel = Channel.create()
myChannel << 'Hello world'
```

5.3 Observing events

5.3.1 subscribe()

The `subscribe()` method permits to execute a user define function each time a new value is emitted by the source channel.

The emitted value is passed implicitly to the specified function. For example:

```
// define a channel emitting three values
source = Channel.from ( 'alpha', 'beta', 'delta' )

// subscribe a function to the channel printing the emitted values
source.subscribe { println "Got: $it" }
```

```
Got: alpha
Got: beta
Got: delta
```

Note: Formally the user defined function is a `Closure` as defined by the Groovy programming language on which the *Nextflow* scripts are based on.

If needed the closure parameter can be defined explicitly, using a name other than `it` and, optionally, specifying the expected value type, as showed in the following example:

```
Channel
  .from( 'alpha', 'beta', 'lambda' )
  .subscribe { String str ->
    println "Got: ${str}; len: ${str.size()}"
  }
```

```
Got: alpha; len: 5
Got: beta; len: 4
Got: lambda; len: 6
```

Read [Closures](#) paragraph to learn more about *closure* feature.

5.3.2 onNext, onComplete, and onError

The `subscribe()` method may accept one or more of the following event handlers:

- `onNext`: registers a function that is invoked whenever the channel emits a value. This is the same as using the `subscribe()` with a *plain* closure as describe in the examples above.

- `onComplete`: registers a function that is invoked after the *last* value is emitted by the channel.
- `onError`: registers a function that it is invoked when an exception is raised while handling the `onNext` event. It will not make further calls to `onNext` or `onComplete`. The `onError` method takes as its parameter the `Throwable` that caused the error.

For example:

```
Channel
  .from( 1, 2, 3 )
  .subscribe onNext: { println it }, onComplete: { println 'Done.' }
```

```
1
2
3
Done.
```


Nextflow *operators* are methods that allow you to connect channels to each other or to transform values emitted by a channel applying some user provided rules.

Operators can be separated in to five groups:

- *Filtering operators*
- *Transforming operators*
- *Splitting operators*
- *Combining operators*
- *Forking operators*
- *Maths operators*
- *Other operators*

6.1 Filtering operators

Given a channel, filtering operators allow you to select only the items that comply with a given rule.

The available filter operators are:

- *distinct*
- *filter*
- *first*
- *last*
- *randomSample*
- *take*
- *unique*

- *until*

6.1.1 filter

The `filter` operator allows you to get only the items emitted by a channel that satisfy a condition and discarding all the others. The filtering condition can be specified by using either a *regular expression*, a literal value, a type *qualifier* (i.e. a Java class) or any boolean *predicate*.

The following example shows how to filter a channel by using a regular expression that returns only string that begins with a:

```
Channel
  .from( 'a', 'b', 'aa', 'bc', 3, 4.5 )
  .filter( ~/^a./ )
  .subscribe { println it }
```

```
a
aa
```

The following example shows how to filter a channel by specifying the type qualifier `Number` so that only numbers are returned:

```
Channel
  .from( 'a', 'b', 'aa', 'bc', 3, 4.5 )
  .filter( Number )
  .subscribe { println it }
```

```
3
4.5
```

Finally, a filtering condition can be defined by using any a boolean *predicate*. A predicate is expressed by a *closure* returning a boolean value. For example the following fragment shows how filter a channel emitting numbers so that the *odd* values are returned:

```
Channel
  .from( 1, 2, 3, 4, 5 )
  .filter { it % 2 == 1 }
  .subscribe { println it }
```

```
1
3
5
```

Tip: In the above example the filter condition is wrapped in curly brackets, instead of round brackets, since it specifies a *closure* as the operator's argument. This just is a language syntax-sugar for `filter({ it.toString().size() == 1 })`

6.1.2 unique

The `unique` operator allows you to remove duplicate items from a channel and only emit single items with no repetition.

For example:

```
Channel
  .from( 1,1,1,5,7,7,7,3,3 )
  .unique()
  .subscribe { println it }
```

```
1
5
7
3
```

You can also specify an optional *closure* that customizes the way it distinguishes between unique items. For example:

```
Channel
  .from(1,3,4,5)
  .unique { it % 2 }
  .subscribe { println it }
```

```
1
4
```

6.1.3 distinct

The `distinct` operator allows you to remove *consecutive* duplicated items from a channel, so that each emitted item is different from the preceding one. For example:

```
Channel
  .from( 1,1,2,2,2,3,1,1,2,2,3 )
  .distinct()
  .subscribe onNext: { println it }, onComplete: { println 'Done' }
```

```
1
2
3
1
2
3
Done
```

You can also specify an optional *closure* that customizes the way it distinguishes between distinct items. For example:

```
Channel
  .from( 1,1,2,2,2,3,1,1,2,4,6 )
  .distinct { it % 2 }
  .subscribe onNext: { println it }, onComplete: { println 'Done' }
```

```
1
2
3
2
Done
```

6.1.4 first

The `first` operator creates a channel that returns the first item emitted by the source channel, or eventually the first item that matches an optional condition. The condition can be specified by using a *regular expression*, a Java *class* type or any boolean *predicate*. For example:

```
// no condition is specified, emits the very first item: 1
Channel
    .from( 1, 2, 3 )
    .first()
    .subscribe { println it }

// emits the first String value: 'a'
Channel
    .from( 1, 2, 'a', 'b', 3 )
    .first( String )
    .subscribe { println it }

// emits the first item matching the regular expression: 'aa'
Channel
    .from( 'a', 'aa', 'aaa' )
    .first( ~/aa.*/ )
    .subscribe { println it }

// emits the first item for which the predicate evaluates to true: 4
Channel
    .from( 1,2,3,4,5 )
    .first { it > 3 }
    .subscribe { println it }
```

6.1.5 randomSample

The `randomSample` operator allows you to create a channel emitting the specified number of items randomly taken from the channel to which is applied. For example:

```
Channel
    .from( 1..100 )
    .randomSample( 10 )
    .println()
```

The above snippet will print 10 numbers in the range from 1 to 100.

The operator supports a second parameter that allows to set the initial *seed* for the random number generator. By setting it, the `randomSample` operator will always return the same pseudo-random sequence. For example:

```
Channel
    .from( 1..100 )
    .randomSample( 10, 234 )
    .println()
```

The above example will print 10 random numbers in the range between 1 and 100. At each run of the script, the same sequence will be returned.

6.1.6 take

The `take` operator allows you to filter only the first n items emitted by a channel. For example:

```
Channel
  .from( 1,2,3,4,5,6 )
  .take( 3 )
  .subscribe onNext: { println it }, onComplete: { println 'Done' }
```

```
1
2
3
Done
```

Note: By specifying the value `-1` the operator takes all values.

See also [until](#).

6.1.7 last

The `last` operator creates a channel that only returns the last item emitted by the source channel. For example:

```
Channel
  .from( 1,2,3,4,5,6 )
  .last()
  .subscribe { println it }
```

```
6
```

6.1.8 until

The `until` operator creates a channel that returns the items emitted by the source channel and stop when the condition specified is verified. For example:

```
Channel
  .from( 3,2,1,5,1,5 )
  .until{ it==5 }
  .println()
```

```
3
2
1
```

See also [take](#).

6.2 Transforming operators

Transforming operators are used to transform the items emitted by a channel to new values.

These operators are:

- *buffer*
- *collate*
- *collect*
- *flatten*
- *flatMap*
- *groupBy*
- *groupTuple*
- *map*
- *reduce*
- *toList*
- *toSortedList*
- *transpose*

6.2.1 map

The `map` operator applies a function of your choosing to every item emitted by a channel, and returns the items so obtained as a new channel. The function applied is called the *mapping* function and is expressed with a *closure* as shown in the example below:

```
Channel
  .from( 1, 2, 3, 4, 5 )
  .map { it * it }
  .subscribe onNext: { println it }, onComplete: { println 'Done' }
```

```
1
4
9
16
25
Done
```

6.2.2 flatMap

The `flatMap` operator applies a function of your choosing to every item emitted by a channel, and returns the items so obtained as a new channel. Whenever the *mapping* function returns a list of items, this list is flattened so that each single item is emitted on its own.

For example:

```
// create a channel of numbers
numbers = Channel.from( 1, 2, 3 )

// map each number to a tuple (array), which items are emitted separately
results = numbers.flatMap { n -> [ n*2, n*3 ] }

// print the final results
results.subscribe onNext: { println it }, onComplete: { println 'Done' }
```

```
2
3
4
6
6
9
Done
```

Associative arrays are handled in the same way, so that each array entry is emitted as a single *key-value* item. For example:

```
Channel.from( 1, 2, 3 )
    .flatMap { it -> [ number: it, square: it*it ] }
    .subscribe { println it.key + ': ' + it.value }
```

```
number: 1
square: 1
number: 2
square: 4
number: 3
square: 9
```

6.2.3 reduce

The `reduce` operator applies a function of your choosing to every item emitted by a channel. Each time this function is invoked it takes two parameters: firstly the *i-th* emitted item and secondly the result of the previous invocation of the function itself. The result is passed on to the next function call, along with the *i+1 th* item, until all the items are processed.

Finally, the `reduce` operator emits the result of the last invocation of your function as the sole output.

For example:

```
Channel
    .from( 1, 2, 3, 4, 5 )
    .reduce { a, b -> println "a: $a b: $b"; return a+b }
    .subscribe { println "result = $it" }
```

It prints the following output:

```
a: 1    b: 2
a: 3    b: 3
a: 6    b: 4
a: 10   b: 5
result = 15
```

Note: In a common usage scenario the first function parameter is used as an *accumulator* and the second parameter represents the *i-th* item to be processed.

Optionally you can specify a *seed* value in order to initialise the accumulator parameter as shown below:

```
myChannel.reduce( seedValue ) { a, b -> ... }
```

6.2.4 groupBy

The `groupBy` operator collects the values emitted by the source channel grouping them together using a *mapping* function that associates each item with a key. When finished, it emits an associative array that maps each key to the set of items identified by that key.

For example:

```
Channel
  .from('hello', 'ciao', 'hola', 'hi', 'bonjour')
  .groupBy { String str -> str[0] }
  .subscribe { println it }
```

```
[ b:['bonjour'], c:['ciao'], h:['hello','hola','hi'] ]
```

The *mapping* function is an optional parameter. When omitted the values are grouped following these rules:

- Any value of type `Map` is associated with the value of its first entry, or `null` when the map itself is empty.
- Any value of type `Map.Entry` is associated with the value of its key attribute.
- Any value of type `Collection` or `Array` is associated with its first entry.
- For any other value, the value itself is used as a key.

6.2.5 groupTuple

The `groupTuple` operator collects tuples (or lists) of values emitted by the source channel grouping together the elements that share the same key. Finally it emits a new tuple object for each distinct key collected.

In other words transform a sequence of tuple like (K, V, W, \dots) into a new channel emitting a sequence of $(K, list(V), list(W), \dots)$

For example:

```
Channel
  .from( [1,'A'], [1,'B'], [2,'C'], [3, 'B'], [1,'C'], [2, 'A'], [3, 'D'] )
  .groupTuple()
  .subscribe { println it }
```

It prints:

```
[1, [A, B, C]]
[2, [C, A]]
[3, [B, D]]
```

By default the first entry in the tuple is used as the grouping key. A different key can be chosen by using the `by` parameter and specifying the index of entry to be used as key (the index is zero-based). For example:

```
Channel
  .from( [1,'A'], [1,'B'], [2,'C'], [3, 'B'], [1,'C'], [2, 'A'], [3, 'D'] )
  .groupTuple(by: 1)
  .subscribe { println it }
```

Grouping by the second value in each tuple the result is:

```
[[1, 2], A]
[[1, 3], B]
[[2, 1], C]
[[3], D]
```

Available parameters:

| Field | Description |
|-------------|---|
| by | The index (zero based) of the element to be used as grouping key. A key composed by multiple elements can be defined specifying a list of indices e.g. <code>by: [0, 2]</code> |
| sort | Defines the sorting criteria for the grouped items. See below for available sorting options. |
| size | The number of items the grouped list(s) has to contain. When the specified size is reached, the tuple is emitted. |
| re-main-der | When <code>false</code> incomplete tuples (i.e. with less than <code>size</code> grouped items) are discarded (default). When <code>true</code> incomplete tuples are emitted as the ending emission. Only valid when a <code>size</code> parameter is specified. |

Sorting options:

| Sort | Description |
|--------|--|
| false | No sorting is applied (default). |
| true | Order the grouped items by the item natural ordering i.e. numerical for number, lexicographic for string, etc. See http://docs.oracle.com/javase/tutorial/collections/interfaces/order.html |
| hash | Order the grouped items by the hash number associated to each entry. |
| deep | Similar to the previous, but the hash number is created on actual entries content e.g. when the item is a file the hash is created on the actual file content. |
| custom | A custom sorting criteria can be specified by using either a <i>Closure</i> or a <i>Comparator</i> object. |

6.2.6 buffer

The `buffer` operator gathers the items emitted by the source channel into subsets and emits these subsets separately.

There are a number of ways you can regulate how `buffer` gathers the items from the source channel into subsets:

- `buffer(closingCondition)`: starts to collect the items emitted by the channel into a subset until the *closing condition* is verified. After that the subset is emitted to the resulting channel and new items are gathered into a new subset. The process is repeated until the last value in the source channel is sent. The `closingCondition` can be specified either as a *regular expression*, a Java class, a literal value, or a *boolean predicate* that has to be satisfied. For example:

```
Channel
    .from( 1,2,3,1,2,3 )
    .buffer { it == 2 }
    .subscribe { println it }

// emitted values
[1,2]
[3,1,2]
```

- `buffer(openingCondition, closingCondition)`: starts to gather the items emitted by the channel as soon as one of them verify the *opening condition* and it continues until there is one item which verify the *closing condition*. After that the subset is emitted and it continues applying the described logic until

the last channel item is emitted. Both conditions can be defined either as a *regular expression*, a literal value, a Java class, or a *boolean predicate* that need to be satisfied. For example:

```
Channel
  .from( 1,2,3,4,5,1,2,3,4,5,1,2 )
  .buffer( 2, 4 )
  .subscribe { println it }

// emits bundles starting with '2' and ending with '4'
[2,3,4]
[2,3,4]
```

- `buffer(size: n)`: transform the source channel in such a way that it emits tuples made up of `n` elements. An incomplete tuple is discarded. For example:

```
Channel
  .from( 1,2,3,1,2,3,1 )
  .buffer( size: 2 )
  .subscribe { println it }

// emitted values
[1, 2]
[3, 1]
[2, 3]
```

If you want to emit the last items in a tuple containing less than `n` elements, simply add the parameter `remainder` specifying `true`, for example:

```
Channel
  .from( 1,2,3,1,2,3,1 )
  .buffer( size: 2, remainder: true )
  .subscribe { println it }

// emitted values
[1, 2]
[3, 1]
[2, 3]
[1]
```

- `buffer(size: n, skip: m)`: as in the previous example, it emits tuples containing `n` elements, but skips `m` values before starting to collect the values for the next tuple (including the first emission). For example:

```
Channel
  .from( 1,2,3,4,5,1,2,3,4,5,1,2 )
  .buffer( size:3, skip:2 )
  .subscribe { println it }

// emitted values
[3, 4, 5]
[3, 4, 5]
```

If you want to emit the remaining items in a tuple containing less than `n` elements, simply add the parameter `remainder` specifying `true`, as shown in the previous example.

See also: [collate](#) operator.

6.2.7 collate

The `collate` operator transforms a channel in such a way that the emitted values are grouped in tuples containing *n* items. For example:

```
Channel
  .from(1,2,3,1,2,3,1)
  .collate( 3 )
  .subscribe { println it }
```

```
[1, 2, 3]
[1, 2, 3]
[1]
```

As shown in the above example the last tuple may be incomplete e.g. contain less elements than the specified size. If you want to avoid this, specify `false` as the second parameter. For example:

```
Channel
  .from(1,2,3,1,2,3,1)
  .collate( 3, false )
  .subscribe { println it }
```

```
[1, 2, 3]
[1, 2, 3]
```

A second version of the `collate` operator allows you to specify, after the *size*, the *step* by which elements are collected in tuples. For example:

```
Channel
  .from(1,2,3,4)
  .collate( 3, 1 )
  .subscribe { println it }
```

```
[1, 2, 3]
[2, 3, 4]
[3, 4]
[4]
```

As before, if you don't want to emit the last items which do not complete a tuple, specify `false` as the third parameter.

See also: *buffer* operator.

6.2.8 collect

The `collect` operator collects all the items emitted by a channel to a `List` and return the resulting object as a sole emission. For example:

```
Channel
  .from( 1, 2, 3, 4 )
  .collect()
  .println()
```

```
# outputs
[1,2,3,4]
```

An optional *closure* can be specified to transform each item before adding it to the resulting list. For example:

```
Channel
  .from( 'hello', 'ciao', 'bonjour' )
  .collect { it.length() }
  .println()

# outputs
[5,4,7]
```

See also: *toList* and *toSortedList* operator.

6.2.9 flatten

The `flatten` operator transforms a channel in such a way that every item of type `Collection` or `Array` is flattened so that each single entry is emitted separately by the resulting channel. For example:

```
Channel
  .from( [1,[2,3]], 4, [5,[6]] )
  .flatten()
  .subscribe { println it }
```

```
1
2
3
4
5
6
```

See also: *flatMap* operator.

6.2.10 toList

The `toList` operator collects all the items emitted by a channel to a `List` object and emits the resulting collection as a single item. For example:

```
Channel
  .from( 1, 2, 3, 4 )
  .toList()
  .subscribe onNext: { println it }, onComplete: 'Done'
```

```
[1,2,3,4]
Done
```

See also: *collect* operator.

6.2.11 toSortedList

The `toSortedList` operator collects all the items emitted by a channel to a `List` object where they are sorted and emits the resulting collection as a single item. For example:

```
Channel
  .from( 3, 2, 1, 4 )
  .toSortedList()
  .subscribe onNext: { println it }, onComplete: 'Done'
```



```
[1, 2, 3, 4]
Done
```

You may also pass a comparator closure as an argument to the `toSortedList` operator to customize the sorting criteria. For example, to sort by the second element of a tuple in descending order:

```
Channel
  .from( ["homer", 5], ["bart", 2], ["lisa", 10], ["marge", 3], ["maggie", 7])
  .toSortedList( { a, b -> b[1] <=> a[1] } )
  .view()
```

```
[[lisa, 10], [maggie, 7], [homer, 5], [marge, 3], [bart, 2]]
```

See also: [collect](#) operator.

6.2.12 transpose

The `transpose` operator transforms a channel in such a way that the emitted items are the result of a transposition of all tuple elements in each item. For example:

```
Channel.from([
  ['a', ['p', 'q'], ['u', 'v']],
  ['b', ['s', 't'], ['x', 'y']]
])
  .transpose()
  .println()
```

The above snippet prints:

```
[a, p, u]
[a, q, v]
[b, s, x]
[b, t, y]
```

Available parameters:

| Field | Description |
|------------|---|
| by | The index (zero based) of the element to be transposed. Multiple elements can be defined specifying as list of indices e.g. <code>by: [0, 2]</code> |
| re-mainder | When <code>false</code> incomplete tuples are discarded (default). When <code>true</code> incomplete tuples are emitted containing a <code>null</code> in place of a missing element. |

6.3 Splitting operators

These operators are used to split items emitted by channels into chunks that can be processed by downstream operators or processes.

The available splitting operators are:

- [splitCsv](#)
- [splitFasta](#)

- *splitFastq*
- *splitText*

6.3.1 splitCsv

The `splitCsv` operator allows you to parse text items emitted by a channel, that are formatted using the [CSV format](#), and split them into records or group them into list of records with a specified length.

In the simplest case just apply the `splitCsv` operator to a channel emitting a CSV formatted text files or text entries. For example:

```
Channel
  .from( 'alpha,beta,gamma\n10,20,30\n70,80,90' )
  .splitCsv()
  .subscribe { row ->
    println "${row[0]} - ${row[1]} - ${row[2]}"
  }
```

The above example shows hows CSV text is parsed and is split into single rows. Values can be accessed by its column index in the row object.

When the CVS begins with a header line defining the columns names, you can specify the parameter `header: true` which allows you to reference each value by its name, as shown in the following example:

```
Channel
  .from( 'alpha,beta,gamma\n10,20,30\n70,80,90' )
  .splitCsv(header: true)
  .subscribe { row ->
    println "${row.alpha} - ${row.beta} - ${row.gamma}"
  }
```

It will print

```
10 - 20 - 30
70 - 80 - 90
```

Alternatively you can provide custom header names by specifying a the list of strings in the `header` parameter as shown below:

```
Channel
  .from( 'alpha,beta,gamma\n10,20,30\n70,80,90' )
  .splitCsv(header: ['col1', 'col2', 'col3'], skip: 1 )
  .subscribe { row ->
    println "${row.col1} - ${row.col2} - ${row.col3}"
  }
```

Available parameters:

| Field | Description |
|------------|---|
| by | The number of rows in each <i>chunk</i> |
| sep | The character used to separate the values (default: ,) |
| quote | Values may be quoted by single or double quote characters. |
| header | When <code>true</code> the first line is used as columns names. Alternatively it can be used to provide the list of columns names. |
| charset | Parse the content by using the specified charset e.g. UTF-8 |
| strip | Removes leading and trailing blanks from values (default: <code>false</code>) |
| skip | Number of lines since the file beginning to ignore when parsing the CSV content. |
| limit | Limits the number of retrieved records for each file to the specified value. |
| decompress | When <code>true</code> decompress the content using the GZIP format before processing it (note: files whose name ends with <code>.gz</code> extension are decompressed automatically) |
| elem | The index of the element to split when the operator is applied to a channel emitting list/tuple objects (default: first file object or first element) |

6.3.2 splitFasta

The `splitFasta` operator allows you to split the entries emitted by a channel, that are formatted using the [FASTA format](#). It returns a channel which emits text item for each sequence in the received FASTA content.

The number of sequences in each text chunk produced by the `splitFasta` operator can be set by using the `by` parameter. The following example shows how to read a FASTA file and split it into chunks containing 10 sequences each:

```
Channel
  .fromPath('misc/sample.fa')
  .splitFasta( by: 10 )
  .subscribe { print it }
```

Warning: By default chunks are kept in memory. When splitting big files specify the parameter `file: true` to save the chunks into files in order to not incur in a `OutOfMemoryException`. See the available parameter table below for details.

A second version of the `splitFasta` operator allows you to split a FASTA content into record objects, instead of text chunks. A record object contains a set of fields that let you access and manipulate the FASTA sequence information with ease.

In order to split a FASTA content into record objects, simply use the `record` parameter specifying the map of required the fields, as shown in the example below:

```
Channel
  .fromPath('misc/sample.fa')
  .splitFasta( record: [id: true, seqString: true] )
  .filter { record -> record.id =~ /^ENST0.* / }
  .subscribe { record -> println record.seqString }
```

Note: In this example, the file `misc/sample.fa` is split into records containing the `id` and the `seqString` fields (i.e. the sequence id and the sequence data). The following `filter` operator only keeps the sequences which ID starts with the `ENST0` prefix, finally the sequence content is printed by using the `subscribe` operator.

Available parameters:

| Field | Description |
|------------|--|
| by | Defines the number of sequences in each <i>chunk</i> (default: 1) |
| size | Defines the size in memory units of the expected chunks eg. <i>1.MB</i> . |
| limit | Limits the number of retrieved sequences for each file to the specified value. |
| record | Parse each entry in the FASTA file as record objects (see following table for accepted values) |
| charset | Parse the content by using the specified charset e.g. UTF-8 |
| compress | When <code>true</code> resulting file chunks are GZIP compressed. The <code>.gz</code> suffix is automatically added to chunk file names. |
| decompress | When <code>true</code> , decompress the content using the GZIP format before processing it (note: files whose name ends with <code>.gz</code> extension are decompressed automatically) |
| file | When <code>true</code> saves each split to a file. Use a string instead of <code>true</code> value to create split files with a specific name (split index number is automatically added). Finally, set this attribute to an existing directory, in order to save the split files into the specified folder. |
| elem | The index of the element to split when the operator is applied to a channel emitting list/tuple objects (default: first file object or first element) |

The following fields are available when using the `record` parameter:

| Field | Description |
|-----------|---|
| id | The FASTA sequence identifier i.e. the word following the <code>></code> symbol up to the first <i>blank</i> or <i>newline</i> character |
| header | The first line in a FASTA sequence without the <code>></code> character |
| desc | The text in the FASTA header following the ID value |
| text | The complete FASTA sequence including the header |
| seqString | The sequence data as a single line string i.e. containing no <i>newline</i> characters |
| sequence | The sequence data as a multi-line string (always ending with a <i>newline</i> character) |
| width | Define the length of a single line when the <code>sequence</code> field is used, after that the sequence data continues on a new line. |

6.3.3 splitFastq

The `splitFastq` operator allows you to split the entries emitted by a channel, that are formatted using the [FASTQ format](#). It returns a channel which emits a text chunk for each sequence in the received item.

The number of sequences in each text chunk produced by the `splitFastq` operator is defined by the parameter `by`. The following example shows you how to read a FASTQ file and split it into chunks containing 10 sequences each:

```
Channel
  .fromPath('misc/sample.fastq')
  .splitFastq( by: 10 )
  .println()
```

Warning: By default chunks are kept in memory. When splitting big files specify the parameter `file: true` to save the chunks into files in order to not incur in a `OutOfMemoryException`. See the available parameter table below for details.

A second version of the `splitFastq` operator allows you to split a FASTQ formatted content into record objects,

instead of text chunks. A record object contains a set of fields that let you access and manipulate the FASTQ sequence data with ease.

In order to split FASTQ sequences into record objects simply use the `record` parameter specifying the map of the required fields, or just specify `record: true` as in the example shown below:

```
Channel
  .fromPath('misc/sample.fastq')
  .splitFastq( record: true )
  .println { record -> record.readHeader }
```

Finally the `splitFastq` operator is able to split paired-end read pair FASTQ files. It must be applied to a channel which emits tuples containing at list two elements that are the files to be splitted. For example:

```
Channel
  .fromFilePairs('/my/data/SRR*_{1,2}.fastq', flat:true)
  .splitFastq(by: 100_000, pe:true, file:true)
  .println()
```

Note: The `fromFilePairs` requires the `flat:true` option to have the file pairs as separate elements in the produced tuples.

Warning: This operator assumes that the order of the PE reads correspond with each other and both files contain the same number of reads.

Available parameters:

| Field | Description |
|-------------------------|--|
| <code>by</code> | Defines the number of <i>reads</i> in each <i>chunk</i> (default: 1) |
| <code>pe</code> | When <code>true</code> splits paired-end read files, therefore items emitted by the source channel must be tuples in which at least two elements are the read-pair files to be splitted. |
| <code>limit</code> | Limits the number of retrieved <i>reads</i> for each file to the specified value. |
| <code>record</code> | Parse each entry in the FASTQ file as record objects (see following table for accepted values) |
| <code>charset</code> | Parse the content by using the specified charset e.g. UTF-8 |
| <code>compress</code> | When <code>true</code> resulting file chunks are GZIP compressed. The <code>.gz</code> suffix is automatically added to chunk file names. |
| <code>decompress</code> | When <code>true</code> decompress the content using the GZIP format before processing it (note: files whose name ends with <code>.gz</code> extension are decompressed automatically) |
| <code>file</code> | When <code>true</code> saves each split to a file. Use a string instead of <code>true</code> value to create split files with a specific name (split index number is automatically added). Finally, set this attribute to an existing directory, in order to save the split files into the specified folder. |
| <code>elem</code> | The index of the element to split when the operator is applied to a channel emitting list/tuple objects (default: first file object or first element) |

The following fields are available when using the `record` parameter:

| Field | Description |
|---------------|--|
| readHeader | Sequence header (without the @ prefix) |
| readString | The raw sequence data |
| qualityHeader | Base quality header (it may be empty) |
| qualityString | Quality values for the sequence |

6.3.4 splitText

The `splitText` operator allows you to split multi-line strings or text file items, emitted by a source channel into chunks containing *n* lines, which will be emitted by the resulting channel.

For example:

```
Channel
  .fromPath('/some/path/*.txt')
  .splitText()
  .subscribe { print it }
```

It splits the content of the files with suffix `.txt`, and prints it line by line.

By default the `splitText` operator splits each item into chunks of one line. You can define the number of lines in each chunk by using the parameter `by`, as shown in the following example:

```
Channel
  .fromPath('/some/path/*.txt')
  .splitText( by: 10 )
  .subscribe {
    print it;
    print "--- end of the chunk ---\n"
  }
```

An optional *closure* can be specified in order to *transform* the text chunks produced by the operator. The following example shows how to split text files into chunks of 10 lines and transform them to capital letters:

```
Channel
  .fromPath('/some/path/*.txt')
  .splitText( by: 10 ) { it.toUpperCase() }
  .subscribe { print it }
```

Note: Text chunks returned by the operator `splitText` are always terminated by a newline character.

Available parameters:

| Field | Description |
|------------|--|
| by | Defines the number of lines in each <i>chunk</i> (default: 1) |
| limit | Limits the number of retrieved lines for each file to the specified value. |
| charset | Parse the content by using the specified charset e.g. UTF-8 |
| compress | When <code>true</code> resulting file chunks are GZIP compressed. The <code>.gz</code> suffix is automatically added to chunk file names. |
| decompress | When <code>true</code> , decompress the content using the GZIP format before processing it (note: files whose name ends with <code>.gz</code> extension are decompressed automatically) |
| file | When <code>true</code> saves each split to a file. Use a string instead of <code>true</code> value to create split files with a specific name (split index number is automatically added). Finally, set this attribute to an existing directory, in order to save the split files into the specified folder. |
| elem | The index of the element to split when the operator is applied to a channel emitting list/tuple objects (default: first file object or first element) |

6.4 Combining operators

The combining operators are:

- *cross*
- *collectFile*
- *combine*
- *concat*
- *into*
- *join*
- *merge*
- *mix*
- *phase*
- *spread*
- *tap*

6.4.1 into

The `into` operator connects a source channel to two or more target channels in such a way the values emitted by the source channel are copied to the target channels. For example:

```
Channel
    .from( 'a', 'b', 'c' )
    .into{ foo; bar }

foo.println{ "Foo emit: " + it }
bar.println{ "Bar emit: " + it }
```

```
Foo emit: a
Foo emit: b
Foo emit: c
Bar emit: a
Bar emit: b
Bar emit: c
```

Note: Note the use in this example of curly brackets and the `;` as channel names separator. This is needed because the actual parameter of `into` is a *closure* which defines the target channels to which the source one is connected.

A second version of the `into` operator takes an integer *n* as an argument and returns a list of *n* channels, each of which emits a copy of the items that were emitted by the source channel. For example:

```
(foo, bar) = Channel.from( 'a', 'b', 'c' ).into(2)
foo.println{ "Foo emit: " + it }
bar.println{ "Bar emit: " + it }
```

Note: The above example takes advantage of the *multiple assignment* syntax in order to assign two variables at once using the list of channels returned by the `into` operator.

See also *tap* and *separate* operators.

6.4.2 tap

The `tap` operator combines the functions of *into* and *separate* operators in such a way that it connects two channels, copying the values from the source into the *tapped* channel. At the same time it splits the source channel into a newly created channel that is returned by the operator itself.

The `tap` can be useful in certain scenarios where you may be required to concatenate multiple operations, as in the following example:

```
log1 = Channel.create().subscribe { println "Log 1: $it" }
log2 = Channel.create().subscribe { println "Log 2: $it" }

Channel
    .from( 'a', 'b', 'c' )
    .tap( log1 )
    .map { it * 2 }
    .tap( log2 )
    .subscribe { println "Result: $it" }
```

```
Log 1: a
Log 1: b
Log 1: c

Log 2: aa
Log 2: bb
Log 2: cc

Result: aa
Result: bb
Result: cc
```


The `tap` operator also allows the target channel to be specified by using a closure. The advantage of this syntax is that you won't need to previously create the target channel, because it is created implicitly by the operator itself.

Using the closure syntax the above example can be rewritten as shown below:

```
Channel
    .from ( 'a', 'b', 'c' )
    .tap { log1 }
    .map { it * 2 }
    .tap { log2 }
    .subscribe { println "Result: $it" }

log1.subscribe { println "Log 1: $it" }
log2.subscribe { println "Log 2: $it" }
```

See also *into* and *separate* operators.

6.4.3 join

The `join` operator creates a channel that joins together the items emitted by two channels for which exists a matching key. The key is defined, by default, as the first element in each item emitted.

For example:

```
left = Channel.from(['X', 1], ['Y', 2], ['Z', 3], ['P', 7])
right= Channel.from(['Z', 6], ['Y', 5], ['X', 4])
left.join(right).println()
```

The resulting channel emits:

```
[Z, 3, 6]
[Y, 2, 5]
[X, 1, 4]
```

The *index* of a different matching element can be specified by using the *by* parameter.

The `join` operator can emit all the pairs that are incomplete, i.e. the items for which a matching element is missing, by specifying the optional parameter *remainder* as shown below:

```
left = Channel.from(['X', 1], ['Y', 2], ['Z', 3], ['P', 7])
right= Channel.from(['Z', 6], ['Y', 5], ['X', 4])
left.join(right, remainder: true).println()
```

The above example prints:

```
[Y, 2, 5]
[Z, 3, 6]
[X, 1, 4]
[P, 7, null]
```

The following parameters can be used with the `join` operator:

| Name | Description |
|-------------|---|
| by | The index (zero based) of the element to be used as grouping key. A key composed by multiple elements can be defined specifying a list of indices e.g. <code>by: [0, 2]</code> |
| re-main-der | When <code>false</code> incomplete tuples (i.e. with less than <i>size</i> grouped items) are discarded (default). When <code>true</code> incomplete tuples are emitted as the ending emission. |

6.4.4 merge

The `merge` operator lets you join items emitted by two (or more) channels into a new channel.

For example the following code merges two channels together, one which emits a series of odd integers and the other which emits a series of even integers:

```
odds  = Channel.from([1, 3, 5, 7, 9]);
evens = Channel.from([2, 4, 6]);

odds
    .merge( evens )
    .println()
```

```
[1, 2]
[3, 4]
[5, 6]
```

An option closure can be provide to customise the items emitted by the resulting merged channel. For example:

```
odds  = Channel.from([1, 3, 5, 7, 9]);
evens = Channel.from([2, 4, 6]);

odds
    .merge( evens ) { a, b -> tuple(b*b, a) }
    .println()
```

6.4.5 mix

The `mix` operator combines the items emitted by two (or more) channels into a single channel.

For example:

```
c1 = Channel.from( 1,2,3 )
c2 = Channel.from( 'a','b' )
c3 = Channel.from( 'z' )

c1 .mix(c2,c3)
    .subscribe onNext: { println it }, onComplete: { println 'Done' }
```

```
1
2
3
'a'
'b'
'z'
```

Note: The items emitted by the resulting mixed channel may appear in any order, regardless of which source channel they came from. Thus, the following example it could be a possible result of the above example as well.

```
'z'
1
'a'
2
'b'
3
```

6.4.6 phase

Warning: This operator is deprecated. Use the *join* operator instead.

The `phase` operator creates a channel that synchronizes the values emitted by two other channels, in such a way that it emits pairs of items that have a matching key.

The key is defined, by default, as the first entry in an array, a list or map object, or the value itself for any other data type.

For example:

```
ch1 = Channel.from( 1,2,3 )
ch2 = Channel.from( 1,0,0,2,7,8,9,3 )
ch1 .phase(ch2) .subscribe { println it }
```

It prints:

```
[1,1]
[2,2]
[3,3]
```

Optionally, a mapping function can be specified in order to provide a custom rule to associate an item to a key, as shown in the following example:

```
ch1 = Channel.from( [sequence: 'aaaaaa', id: 1], [sequence: 'bbbbbb', id: 2] )
ch2 = Channel.from( [val: 'zzzz', id: 3], [val: 'xxxxx', id: 1], [val: 'yyyyy', id: 2] )
ch1 .phase(ch2) { it -> it.id } .subscribe { println it }
```

It prints:

```
[[sequence:aaaaaa, id:1], [val:xxxxx, id:1]]
[[sequence:bbbbbb, id:2], [val:yyyyy, id:2]]
```

Finally, the `phase` operator can emit all the pairs that are incomplete, i.e. the items for which a matching element is missing, by specifying the optional parameter `remainder` as shown below:

```
ch1 = Channel.from( 1,0,0,2,5,3 )
ch2 = Channel.from( 1,2,3,4 )
ch1 .phase(ch2, remainder: true) .subscribe { println it }
```

It prints:

```
[1, 1]
[2, 2]
[3, 3]
[0, null]
[0, null]
[5, null]
[null, 4]
```

See also [join](#) operator.

6.4.7 cross

The `cross` operators allows you to combine the items of two channels in such a way that the items of the source channel are emitted along with the items emitted by the target channel for which they have a matching key.

The key is defined, by default, as the first entry in an array, a list or map object, or the value itself for any other data type. For example:

```
source = Channel.from( [1, 'alpha'], [2, 'beta'] )
target = Channel.from( [1, 'x'], [1, 'y'], [1, 'z'], [2, 'p'], [2, 'q'], [2, 't'] )

source.cross(target).subscribe { println it }
```

It will output:

```
[ [1, alpha], [1, x] ]
[ [1, alpha], [1, y] ]
[ [1, alpha], [1, z] ]
[ [2, beta], [2, p] ]
[ [2, beta], [2, q] ]
[ [2, beta], [2, t] ]
```

The above example shows how the items emitted by the source channels are associated to the ones emitted by the target channel (on the right) having the same key.

There are two important caveats when using the `cross` operator:

1. The operator is not *reflexive*, i.e. the result of `a.cross(b)` is different from `b.cross(a)`
2. The source channel should emits items for which there's no key repetition i.e. the emitted items have an unique key identifier.

Optionally, a mapping function can be specified in order to provide a custom rule to associate an item to a key, in a similar manner as shown for the [phase](#) operator.

6.4.8 collectFile

The `collectFile` operator allows you to gather the items emitted by a channel and save them to one or more files. The operator returns a new channel that emits the collected file(s).

In the simplest case, just specify the name of a file where the entries have to be stored. For example:

```
Channel
  .from('alpha', 'beta', 'gamma')
  .collectFile(name: 'sample.txt', newLine: true)
  .subscribe {
```

(continues on next page)

(continued from previous page)

```
println "Entries are saved to file: $it"
println "File content is: ${it.text}"
}
```

A second version of the `collectFile` operator allows you to gather the items emitted by a channel and group them together into files whose name can be defined by a dynamic criteria. The grouping criteria is specified by a *closure* that must return a pair in which the first element defines the file name for the group and the second element the actual value to be appended to that file. For example:

```
Channel
  .from('Hola', 'Ciao', 'Hello', 'Bonjour', 'Halo')
  .collectFile() { item ->
    [ "${item[0]}.txt", item + '\n' ]
  }
  .subscribe {
    println "File ${it.name} contains:"
    println it.text
  }
```

It will print:

```
File 'B.txt' contains:
Bonjour

File 'C.txt' contains:
Ciao

File 'H.txt' contains:
Halo
Hola
Hello
```

Tip: When the items emitted by the source channel are files, the grouping criteria can be omitted. In this case the items content will be grouped in file(s) having the same name as the source items.

The following parameters can be used with the `collectFile` operator:

| Name | Description |
|-------------|---|
| keep-Header | Prepend the resulting file with the header fetched in the first collected file. The header size (ie. lines) can be specified by using the <code>size</code> parameter (default: <code>false</code>). |
| name | Name of the file where all received values are stored. |
| new-Line | Appends a <code>newline</code> character automatically after each entry (default: <code>false</code>). |
| seed | A value or a map of values used to initialise the files content. |
| skip | Skip the first <i>n</i> lines eg. <code>skip: 1</code> . |
| sort | Defines sorting criteria of content in resulting file(s). See below for sorting options. |
| storeDir | Folder where the resulting file(s) are be stored. |
| tempDir | Folder where temporary files, used by the collecting process, are stored. |

Note: The file content is sorted in such a way that it does not depend on the order on which entries have been added

to it, this guarantees that it is consistent (i.e. do not change) across different executions with the same data.

The ordering of file's content can be defined by using the `sort` parameter. The following criteria can be specified:

| Sort | Description |
|----------|---|
| false | Disable content sorting. Entries are appended as they are produced. |
| true | Order the content by the entries natural ordering i.e. numerical for number, lexicographic for string, etc. See http://docs.oracle.com/javase/tutorial/collections/interfaces/order.html |
| in-index | Order the content by the incremental index number assigned to each entry while they are collected. |
| hash | Order the content by the hash number associated to each entry (default) |
| deep | Similar to the previous, but the hash number is created on actual entries content e.g. when the entry is a file the hash is created on the actual file content. |
| custom | A custom sorting criteria can be specified by using either a <i>Closure</i> or a <i>Comparator</i> object. |

For example the following snippet shows how sort the content of the result file alphabetically:

```
Channel
  .from('Z'..'A')
  .collectFile(name:'result', sort: true, newline: true)
  .subscribe {
    println it.text
  }
```

It will print:

```
A
B
C
:
Z
```

The following example shows how use a *closure* to collect and sort all sequences in a FASTA file from shortest to longest:

```
Channel
  .fromPath('/data/sequences.fa')
  .splitFasta( record: [id: true, sequence: true] )
  .collectFile( name:'result.fa', sort: { it.size() } ) {
    it.sequence
  }
  .subscribe { println it.text }
```

Warning: The `collectFile` operator to carry out its function need to store in a temporary folder that is automatically deleted on job completion. For performance reason this folder is allocated in the machine local storage, and it will require as much free space as are the data you are collecting. Optionally, an alternative temporary data folder can be specified by using the `tempDir` parameter.

6.4.9 combine

The `combine` operator combines (cartesian product) the items emitted by two channels or by a channel and a `Collection` object (as right operand). For example:

```
numbers = Channel.from(1,2,3)
words = Channel.from('hello', 'ciao')
numbers
    .combine(words)
    .println()

# outputs
[1, hello]
[2, hello]
[3, hello]
[1, ciao]
[2, ciao]
[3, ciao]
```

A second version of the `combine` operator allows you to combine between them those items that share a common matching key. The index of the key element is specified by using the `by` parameter (the index is zero-based, multiple indexes can be specified with list a integers). For example:

```
left = Channel.from(['A',1], ['B',2], ['A',3])
right = Channel.from(['B','x'], ['B','y'], ['A','z'], ['A','w'])

left
    .combine(right, by: 0)
    .println()

# outputs
[A, 1, z]
[A, 3, z]
[A, 1, w]
[A, 3, w]
[B, 2, x]
[B, 2, y]
```

See also *cross*, *spread* and *phase*.

6.4.10 concat

The `concat` operator allows you to *concatenate* the items emitted by two or more channels to a new channel, in such a way that the items emitted by the resulting channel are in same order as they were when specified as operator arguments.

In other words it guarantees that given any n channels, the concatenation channel emits the items proceeding from the channel $i+1$ *th* only after *all* the items proceeding from the channel i *th* were emitted.

For example:

```
a = Channel.from('a','b','c')
b = Channel.from(1,2,3)
c = Channel.from('p','q')

c.concat( b, a ).subscribe { println it }
```

It will output:

```
p
q
1
2
3
a
b
c
```

6.4.11 spread

Warning: This operator is deprecated. See [combine](#) instead.

The `spread` operator combines the items emitted by the source channel with all the values in an array or a `Collection` object specified as the operator argument. For example:

```
Channel
  .from(1,2,3)
  .spread(['a','b'])
  .subscribe onNext: { println it }, onComplete: { println 'Done' }
```

```
[1, 'a']
[1, 'b']
[2, 'a']
[2, 'b']
[3, 'a']
[3, 'b']
Done
```

6.5 Forking operators

The forking operators are:

- *choice*
- *separate*
- *route*

6.5.1 choice

The `choice` operator allows you to forward the items emitted by a source channel to two (or more) output channels, *choosing* one out of them at a time.

The destination channel is selected by using a *closure* that must return the *index* number of the channel where the item has to be sent. The first channel is identified by the index 0, the second as 1 and so on.

The following example sends all string items beginning with `Hello` into `queue1`, the others into `queue2`


```
source = Channel.from 'Hello world', 'Hola', 'Hello John'
queue1 = Channel.create()
queue2 = Channel.create()

source.choice( queue1, queue2 ) { a -> a =~ /^Hello.*/ ? 0 : 1 }

queue1.subscribe { println it }
```

6.5.2 separate

The `separate` operator lets you copy the items emitted by the source channel into multiple channels, which each of these can receive a *separate* version of the same item.

The operator applies a *mapping function* of your choosing to every item emitted by the source channel. This function must return a list of as many values as there are output channels. Each entry in the result list will be assigned to the output channel with the corresponding position index. For example:

```
queue1 = Channel.create()
queue2 = Channel.create()

Channel
  .from ( 2,4,8 )
  .separate( queue1, queue2 ) { a -> [a+1, a*a] }

queue1.subscribe { println "Channel 1: $it" }
queue2.subscribe { println "Channel 2: $it" }
```

```
Channel 1: 3
Channel 2: 4
Channel 1: 5
Channel 2: 16
Channel 2: 64
Channel 1: 9
```

When the *mapping function* is omitted, the source channel must emit tuples of values. In this case the operator `separate` splits the tuple in such a way that the value *i-th* in a tuple is assigned to the target channel with the corresponding position index. For example:

```
alpha = Channel.create()
delta = Channel.create()

Channel
  .from([1,2], ['a','b'], ['p','q'])
  .separate( alpha, delta )

alpha.subscribe { println "first : $it" }
delta.subscribe { println "second: $it" }
```

It will output:

```
first : 1
first : a
first : p
second: 2
second: b
second: q
```

A second version of the `separate` operator takes an integer n as an argument and returns a list of n channels, each of which gets a value from the corresponding element in the list returned by the closure as explained above. For example:

```
source = Channel.from(1,2,3)
(queue1, queue2, queue3) = source.separate(3) { a -> [a, a+1, a*a] }

queue1.subscribe { println "Queue 1 > $it" }
queue2.subscribe { println "Queue 2 > $it" }
queue3.subscribe { println "Queue 3 > $it" }
```

The output will look like the following fragment:

```
Queue 1 > 1
Queue 1 > 2
Queue 1 > 3
Queue 2 > 2
Queue 2 > 3
Queue 2 > 4
Queue 3 > 1
Queue 3 > 4
Queue 3 > 9
```

Warning: In the above example, please note that since the `subscribe` operator is asynchronous, the output of `channel2` and `channel3` can be printed before the content of `channel1`.

Note: The above example takes advantage of the *multiple assignment* syntax in order to assign two variables at once using the list of channels returned by the `separate` operator.

See also: *into*, *choice* and *map* operators.

6.5.3 route

Warning: This operator is deprecated. It will be removed in a future release.

The `route` operator allows you to forward the items emitted by the source channel to a channel which is associated with the item's key.

The channel's keys are specified by using a map parameter as the operator argument, that associates each channel with a key identifier.

The item's key is defined, by default, as the first entry in an array, a list or map object, or the value itself for any other data type.

Optionally, a mapping function can be specified as a parameter in order to provide a custom rule to associate an item with a key, as shown in the example below:

```
r1 = Channel.create()
r2 = Channel.create()
r3 = Channel.create()

Channel
```

(continues on next page)

(continued from previous page)

```
.from('hello','ciao','hola','hi','x','bonjour')
.route ( b: r1, c: r2, h: r3 ) { it[0] }

r3.subscribe { println it }
```

```
hello
hola
hi
```

In the above example all the string items starting with the letter `b` are copied to the channel `r1`, the items that begin with `c` to the channel `r2` and the ones beginning with `h` are copied to the channel `r3`. Other items eventually existing are discarded.

See also: [into](#), [choice](#) and [separate](#) operators.

6.6 Maths operators

This section talks about operators that performs maths operations on channels.

The maths operators are:

- [count](#)
- [countBy](#)
- [min](#)
- [max](#)
- [sum](#)
- [toInteger](#)

6.6.1 count

The `count` operator creates a channel that emits a single item: a number that represents the total number of items emitted by the source channel. For example:

```
Channel
  .from(9,1,7,5)
  .count()
  .subscribe { println it }
// -> 4
```

An optional parameter can be provided in order to select which items are to be counted. The selection criteria can be specified either as a [regular expression](#), a literal value, a Java class, or a *boolean predicate* that needs to be satisfied. For example:

```
Channel
  .from(4,1,7,1,1)
  .count(1)
  .subscribe { println it }
// -> 3

Channel
```

(continues on next page)

(continued from previous page)

```
.from('a','c','c','q','b')
.count ( ~/c/ )
.subscribe { println it }
// -> 2

Channel
  .from('a','c','c','q','b')
  .count { it <= 'c' }
  .subscribe { println it }
// -> 4
```

6.6.2 countBy

The `countBy` operator creates a channel which emits an associative array (i.e. Map object) that counts the occurrences of the emitted items in the source channel having the same key. For example:

```
Channel
  .from( 'x', 'y', 'x', 'x', 'z', 'y' )
  .countBy()
  .subscribe { println it }
```

```
[x:3, y:2, z:1]
```

An optional grouping criteria can be specified by using a *closure* that associates each item with the grouping key. For example:

```
Channel
  .from( 'hola', 'hello', 'ciao', 'bonjour', 'halo' )
  .countBy { it[0] }
  .subscribe { println it }
```

```
[h:3, c:1, b:1]
```

6.6.3 min

The `min` operator waits until the source channel completes, and then emits the item that has the lowest value. For example:

```
Channel
  .from( 8, 6, 2, 5 )
  .min()
  .subscribe { println "Min value is $it" }
```

```
Min value is 2
```

An optional *closure* parameter can be specified in order to provide a function that returns the value to be compared. The example below shows how to find the string item that has the minimum length:

```
Channel
  .from("hello","hi","hey")
  .min { it.size() }
  .subscribe { println it }
```

```
"hi"
```

Alternatively it is possible to specify a comparator function i.e. a *closure* taking two parameters that represent two emitted items to be compared. For example:

```
Channel
  .from("hello", "hi", "hey")
  .min { a,b -> a.size() <=> b.size() }
  .subscribe { println it }
```

6.6.4 max

The `max` operator waits until the source channel completes, and then emits the item that has the greatest value. For example:

```
Channel
  .from( 8, 6, 2, 5 )
  .min()
  .subscribe { println "Max value is $it" }
```

```
Max value is 8
```

An optional *closure* parameter can be specified in order to provide a function that returns the value to be compared. The example below shows how to find the string item that has the maximum length:

```
Channel
  .from("hello", "hi", "hey")
  .max { it.size() }
  .subscribe { println it }
```

```
"hello"
```

Alternatively it is possible to specify a comparator function i.e. a *closure* taking two parameters that represent two emitted items to be compared. For example:

```
Channel
  .from("hello", "hi", "hey")
  .max { a,b -> a.size() <=> b.size() }
  .subscribe { println it }
```

6.6.5 sum

The `sum` operator creates a channel that emits the sum of all the items emitted by the channel itself. For example:

```
Channel
  .from( 8, 6, 2, 5 )
  .sum()
  .subscribe { println "The sum is $it" }
```

```
The sum is 21
```

An optional *closure* parameter can be specified in order to provide a function that, given an item, returns the value to be summed. For example:

```
Channel
    .from( 4, 1, 7, 5 )
    .sum { it * it }
    .subscribe { println "Square: $it" }
```

```
Square: 91
```

6.6.6 toInteger

The `toInteger` operator allows you to convert the string values emitted by a channel to `Integer` values. For example:

```
Channel
    .from( '1', '7', '12' )
    .toInteger()
    .sum()
    .println()
```

6.7 Other operators

- *close*
- *dump*
- *ifEmpty*
- *print*
- *println*
- *set*
- *view*

6.7.1 dump

The `dump` operator prints the items emitted by the channel to which is applied only when the option `-dump-channels` is specified on the `run` command line, otherwise it is ignored.

This is useful to enable the debugging of one or more channel content on-demand by using a command line option instead of modifying your script code.

An optional `tag` parameter allows you to select which channel to dump. For example:

```
Channel
    .from(1,2,3)
    .map { it+1 }
    .dump(tag:'foo')
```

```
Channel
    .from(1,2,3)
    .map { it^2 }
    .dump(tag: 'bar')
```

Then you will be able to specify the tag `foo` or `bar` as an argument of the `-dump-channels` option to print either the content of the first or the second channel. Multiple tag names can be specified separating them with a `,` character.

6.7.2 set

The `set` operator assigns the channel to a variable whose name is specified as a closure parameter. For example:

```
Channel.from(10,20,30).set { my_channel }
```

This is semantically equivalent to the following assignment:

```
my_channel = Channel.from(10,20,30)
```

However the `set` operator is more idiomatic in Nextflow scripting, since it can be used at the end of a chain of operator transformations, thus resulting in a more fluent and readable operation.

6.7.3 ifEmpty

The `ifEmpty` operator creates a channel which emits a default value, specified as the operator parameter, when the channel to which is applied is *empty* i.e. doesn't emit any value. Otherwise it will emit the same sequence of entries as the original channel.

Thus, the following example prints:

```
Channel .from(1,2,3) .ifEmpty('Hello') .println()
1
2
3
```

Instead, this one prints:

```
Channel.empty().ifEmpty('Hello') .println()
Hello
```

The `ifEmpty` value parameter can be defined with a *closure*. In this case the result value of the closure evaluation will be emitted when the empty condition is satisfied.

See also: *empty* method.

6.7.4 print

The `print` operator prints the items emitted by a channel to the standard output. An optional *closure* parameter can be specified to customise how items are printed. For example:

```
Channel
  .from('foo', 'bar', 'baz', 'qux')
  .print { it.toUpperCase() + ' ' }
```

It prints:

```
FOO BAR BAZ QUX
```

See also: *println* and *view*.

6.7.5 println

The `println` operator prints the items emitted by a channel to the console standard output appending a *new line* character to each of them. For example:

```
Channel
  .from('foo', 'bar', 'baz', 'qux')
  .println()
```

It prints:

```
foo
bar
baz
qux
```

An optional closure parameter can be specified to customise how items are printed. For example:

```
Channel
  .from('foo', 'bar', 'baz', 'qux')
  .println { "~ $it" }
```

It prints:

```
~ foo
~ bar
~ baz
~ qux
```

See also: [print](#) and [view](#).

6.7.6 view

The `view` operator prints the items emitted by a channel to the console standard output. For example:

```
Channel.from(1,2,3).view()

1
2
3
```

Each item is printed on a separate line unless otherwise specified by using the `newline: false` optional parameter.

How the channel items are printed can be controlled by using an optional closure parameter. The closure it must return the actual value of the item being to be printed:

```
Channel.from(1,2,3)
  .map { it -> [it, it*it] }
  .view { num, sqr -> "Square of: $num is $sqr" }
```

It prints:

```
Square of: 1 is 1
Square of: 2 is 4
Square of: 3 is 9
```

Note: Both the *view* and *print* (or *println*) operators consume the items emitted by the source channel to which they are applied. However, the main difference between them is that the former returns a newly created channel whose content is identical to the source one. This allows the *view* operator to be chained like other operators.

6.7.7 close

The `close` operator sends a termination signal over the channel, causing downstream processes or operators to stop. In a common usage scenario channels are closed automatically by Nextflow, so you won't need to use this operator explicitly.

See also: *empty* factory method.

In the Nextflow framework architecture, the *executor* is the component that determines the system where a pipeline process is run and supervises its execution.

The *executor* provides an abstraction between the pipeline processes and the underlying execution system. This allows you to write the pipeline functional logic independently from the actual processing platform.

In other words you can write your pipeline script once and have it running on your computer, a cluster resource manager or the cloud by simply changing the executor definition in the Nextflow configuration file.

7.1 Local

The *local* executor is used by default. It runs the pipeline processes in the computer where Nextflow is launched. The processes are parallelised by spawning multiple *threads* and by taking advantage of multi-cores architecture provided by the CPU.

In a common usage scenario, the *local* executor can be useful to develop and test your pipeline script in your computer, switching to a cluster facility when you need to run it on production data.

7.2 SGE

The *SGE* executor allows you to run your pipeline script by using a [Sun Grid Engine](#) cluster or a compatible platform ([Open Grid Engine](#), [Univa Grid Engine](#), etc).

Nextflow manages each process as a separate grid job that is submitted to the cluster by using the `qsub` command.

Being so, the pipeline must be launched from a node where the `qsub` command is available, that is, in a common usage scenario, the cluster *head* node.

To enable the SGE executor simply set to `process.executor` property to `sge` value in the `nextflow.config` file.

The amount of resources requested by each job submission is defined by the following process directives:

- *cpus*
- *queue*
- *memory*
- *penv*
- *time*
- *clusterOptions*

7.3 LSF

The *LSF* executor allows you to run your pipeline script by using a [Platform LSF](#) cluster.

Nextflow manages each process as a separate job that is submitted to the cluster by using the `bsub` command.

Being so, the pipeline must be launched from a node where the `bsub` command is available, that is, in a common usage scenario, the cluster *head* node.

To enable the LSF executor simply set to `process.executor` property to `lsf` value in the `nextflow.config` file.

The amount of resources requested by each job submission is defined by the following process directives:

- *cpus*
- *queue*
- *time*
- *memory*
- *clusterOptions*

Note: LSF supports both *per-core* and *per-job* memory limit. Nextflow assumes that LSF works in the *per-core* memory limits mode, thus it divides the requested *memory* by the number of requested *cpus*.

This is not required when LSF is configured to work in *per-job* memory limit mode. You will need to specified that adding the option `perJobMemLimit` in *Scope executor* in the Nextflow configuration file.

See also the [Platform LSF](#) documentation.

7.4 SLURM

The *SLURM* executor allows you to run your pipeline script by using the [SLURM](#) resource manager.

Nextflow manages each process as a separate job that is submitted to the cluster by using the `sbatch` command.

Being so, the pipeline must be launched from a node where the `sbatch` command is available, that is, in a common usage scenario, the cluster *head* node.

To enable the SLURM executor simply set to `process.executor` property to `slurm` value in the `nextflow.config` file.

The amount of resources requested by each job submission is defined by the following process directives:

- *cpus*

- *queue*
- *time*
- *memory*
- *clusterOptions*

Note: SLURM *partitions* can be considered jobs queues. Nextflow allows to set partitions by using the above *queue* directive.

7.5 PBS/Torque

The *PBS* executor allows you to run your pipeline script by using a resource manager belonging to the *PBS/Torque* family of batch schedulers.

Nextflow manages each process as a separate job that is submitted to the cluster by using the `qsub` command provided by the scheduler.

Being so, the pipeline must be launched from a node where the `qsub` command is available, that is, in a common usage scenario, the cluster *login* node.

To enable the PBS executor simply set the property `process.executor = 'pbs'` in the `nextflow.config` file.

The amount of resources requested by each job submission is defined by the following process directives:

- *cpus*
- *queue*
- *time*
- *memory*
- *clusterOptions*

7.6 NQSII

The *NQSII* executor allows you to run your pipeline script by using the *NQSII* resource manager.

Nextflow manages each process as a separate job that is submitted to the cluster by using the `qsub` command provided by the scheduler.

Being so, the pipeline must be launched from a node where the `qsub` command is available, that is, in a common usage scenario, the cluster *login* node.

To enable the NQSII executor simply set the property `process.executor = 'nqsii'` in the `nextflow.config` file.

The amount of resources requested by each job submission is defined by the following process directives:

- *cpus*
- *queue*
- *time*
- *memory*

- *clusterOptions*

7.7 HTCondor

The *HTCondor* executor allows you to run your pipeline script by using the [HTCondor](#) resource manager.

Warning: This is an incubating feature. It may change in future Nextflow releases.

Nextflow manages each process as a separate job that is submitted to the cluster by using the `condor_submit` command.

Being so, the pipeline must be launched from a node where the `condor_submit` command is available, that is, in a common usage scenario, the cluster *head* node.

To enable the HTCondor executor simply set to `process.executor` property to `condor` value in the `nextflow.config` file.

The amount of resources requested by each job submission is defined by the following process directives:

- *cpus*
- *time*
- *memory*
- *disk*
- *clusterOptions*

7.8 Ignite

The *Ignite* executor allows you to run a pipeline by using the [Apache Ignite](#) clustering technology that is embedded with the Nextflow runtime.

To enable this executor set the property `process.executor = 'ignite'` in the `nextflow.config` file.

The amount of resources requested by each task submission is defined by the following process directives:

- *cpus*
- *disk*
- *memory*

Read the [Apache Ignite](#) section in this documentation to learn how to configure Nextflow to deploy and run an Ignite cluster in your infrastructure.

7.9 Kubernetes

Nextflow provides an experimental support for [Kubernetes](#) clustering technology. It allows you to deploy and transparently run a Nextflow pipeline in a Kubernetes cluster.

The following directives can be used to define the amount of computing resources needed and the container(s) to use:

- *cpus*

- *memory*
- *container*

See the [Kubernetes documentation](#) to learn how to deploy a workflow execution in a Kubernetes cluster.

7.10 AWS Batch

Nextflow supports [AWS Batch](#) service which allows submitting jobs in the cloud without having to spin out and manage a cluster of virtual machines. AWS Batch uses Docker containers to run tasks, which makes deploying pipelines much simpler.

The pipeline processes must specify the Docker image to use by defining the `container` directive, either in the pipeline script or the `nextflow.config` file.

To enable this executor set the property `process.executor = 'awsbatch'` in the `nextflow.config` file.

The pipeline can be launched either in a local computer or a EC2 instance. The latter is suggested for heavy or long running workloads. Moreover a S3 bucket must be used as pipeline work directory.

See the [AWS Batch](#) page for further configuration details.

8.1 Configuration file

When a pipeline script is launched Nextflow looks for a file named `nextflow.config` in the current directory and in the script base directory (if it is not the same as the current directory). Finally it checks for the file `$HOME/.nextflow/config`.

When more than one on the above files exist they are merged, so that the settings in the first override the same ones that may appear in the second one, and so on.

The default config file search mechanism can be extended providing an extra configuration file by using the command line option `-c <config file>`.

Note: It's worth noting that by doing this, the files `nextflow.config` and `$HOME/.nextflow/config` are not ignored and they are merged as explained above.

Tip: If you want to ignore any default configuration files and use only the custom one use the command line option `-C <config file>`.

8.1.1 Config syntax

A Nextflow configuration file is a simple text file containing a set of properties defined using the syntax:

```
name = value
```

Please note, string values need to be wrapped in quotation characters while numbers and boolean values (`true`, `false`) do not. Also note that values are typed, meaning for example that, `1` is different from `'1'`, since the first is interpreted as the number one, while the latter is interpreted as a string value.

8.1.2 Config Variables

Configuration properties can be used as variables in the configuration file itself, by using the usual `$propertyName` or `${expression}` syntax.

For example:

```
propertyOne = 'world'
anotherProp = "Hello $propertyOne"
customPath = "$PATH:/my/app/folder"
```

Please note, the usual rules for *String interpolation* are applied, thus a string containing a variable reference must be wrapped in double-quote chars instead of single-quote chars.

The same mechanism allows you to access environment variables defined in the hosting system. Any variable whose name is not defined in the Nextflow configuration file(s) is supposed to be a reference to an environment variable with that name. So, in the above example the property `customPath` is defined as the current system `PATH` to which the string `/my/app/folder` is appended.

Warning: If you are accessing an environment variable that may not exist in the system, your property may contain an undefined value. You can avoid this by using a conditional expression in your property definition as shown below.

```
mySafeProperty = "${MY_FANCY_VARIABLE?:''}"
```

8.1.3 Config comments

Configuration files use the same conventions for comments used by the Groovy or Java programming languages. Thus, use `//` to comment a single line or `/* .. */` to comment a block on multiple lines.

8.1.4 Config include

A configuration file can include one or more configuration files using the keyword `includeConfig`. For example:

```
process.executor = 'sge'
process.queue = 'long'
process.memory = '10G'

includeConfig 'path/foo.config'
```

When a relative path is used, it is resolved against the actual location of the including file.

8.2 Config scopes

Configuration settings can be organized in different scopes by dot prefixing the property names with a scope identifier or grouping the properties in the same scope using the curly brackets notation. This is shown in the following example:

```
alpha.x = 1
alpha.y = 'string value..'
```

(continues on next page)

(continued from previous page)

```
beta {  
    p = 2  
    q = 'another string ..'  
}
```

8.2.1 Scope *env*

The `env` scope allows you to define one or more environment variables that will be exported to the system environment where pipeline processes need to be executed.

Simply prefix your variable names with the `env` scope or surround them by curly brackets, as shown below:

```
env.ALPHA = 'some value'  
env.BETA = "$HOME/some/path"  
  
env {  
    DELTA = 'one more'  
    GAMMA = "/my/path:$PATH"  
}
```

8.2.2 Scope *params*

The `params` scope allows you to define parameters that will be accessible in the pipeline script. Simply prefix the parameter names with the `params` scope or surround them by curly brackets, as shown below:

```
params.custom_param = 123  
params.another_param = 'string value .. '  
  
params {  
    alpha_1 = true  
    beta_2 = 'another string ..'  
}
```

8.2.3 Scope *process*

The `process` configuration scope allows you to provide the default configuration for the processes in your pipeline.

You can specify here any property described in the *process directive* and the `executor` sections. For examples:

```
process {  
    executor='sge'  
    queue='long'  
    clusterOptions = '-pe smp 10 -l virtual_free=64G,h_rt=30:00:00'  
}
```

By using this configuration all processes in your pipeline will be executed through the SGE cluster, with the specified settings.

Process selectors

The `withLabel` selectors allow the configuration of all processes annotated with a *label* directive as shown below:

```
process {
  withLabel: big_mem {
    cpus = 16
    memory = 64.GB
    queue = 'long'
  }
}
```

The above configuration example assigns 16 cpus, 64 Gb of memory and the `long` queue to all processes annotated with the `big_mem` label.

In the same manner, the `withName` selector allows the configuration of a specific process in your pipeline by its name. For example:

```
process {
  withName: hello {
    cpus = 4
    memory = 8.GB
    queue = 'short'
  }
}
```

Tip: Either label and process names do not need to be enclosed with quote characters, provided the name does include special characters (e.g. `-`, `!`, etc) or it's not a keyword or a built-in type identifier. In case of doubt, you can enclose the label names or the process names with single or double quote characters.

Selector expressions

Both label and process name selectors allow the use of a regular expression in order to apply the same configuration to all processes matching the specified pattern condition. For example:

```
process {
  withLabel: 'foo|bar' {
    cpus = 2
    memory = 4.GB
  }
}
```

The above configuration snippet sets 2 cpus and 4 GB of memory to the processes annotated with with a label `foo` and `bar`.

A process selector can be negated prefixing it with the special character `!`. For example:

```
process {
  withLabel: 'foo' { cpus = 2 }
  withLabel: '!foo' { cpus = 4 }
  withName: '!align.*' { queue = 'long' }
}
```

The above configuration snippet sets 2 cpus for the processes annotated with the `foo` label and 4 cpus to all processes *not* annotated with that label. Finally it sets the use of `long` queue to all process whose name does *not* start with

align.

Selectors priority

When mixing generic process configuration and selectors the following priority rules are applied (from lower to higher):

1. Process generic configuration.
2. Process specific directive defined in the workflow script.
3. `withLabel` selector definition.
4. `withName` selector definition.

For example:

```
process {  
  cpus = 4  
  withLabel: foo { cpus = 8 }  
  withName: bar { cpus = 32 }  
}
```

Using the above configuration snippet, all workflow processes use 4 cpus if not otherwise specified in the workflow script. Moreover processes annotated with the `foo` label use 8 cpus. Finally the process named `bar` uses 32 cpus.

8.2.4 Scope *executor*

The `executor` configuration scope allows you to set the optional executor settings, listed in the following table.

| Name | Description |
|-------------------|---|
| name | The name of the executor to be used e.g. <code>local</code> , <code>sge</code> , etc. |
| queueSize | The number of tasks the executor will handle in a parallel manner (default: 100). |
| pollInterval | Determines how often a poll occurs to check for a process termination. |
| dumpInterval | Determines how often the executor status is written in the application log file (default: 5min). |
| queueStatInterval | Determines how often the queue status is fetched from the cluster system. This setting is used only by grid executors (default: 1min). |
| exitReadTimeout | Determines how long the executor waits before return an error status when a process is terminated but the <code>exit</code> file does not exist or it is empty. This setting is used only by grid executors (default: 270 sec). |
| killBatchSize | Determines the number of jobs that can be <i>killed</i> in a single command execution (default: 100). |
| submitRateLimit | Determines the max rate of jobs that can be executed per time unit, for example <code>'10 sec'</code> eg. max 10 jobs per second (default: <i>unlimited</i>). |
| perJobMemLimit | Specifies Platform LSF <i>per-job</i> memory limit mode. See LSF . |
| jobName | Determines the name of jobs submitted to the underlying cluster executor e.g. <code>executor.jobName = { "\$task.name - \$task.hash" }</code> . |
| cpus | The maximum number of CPUs made available by the underlying system (only used by the <code>local</code> executor). |
| memory | The maximum amount of memory made available by the underlying system (only used by the <code>local</code> executor). |

The executor settings can be defined as shown below:

```
executor {
  name = 'sge'
  queueSize = 200
  pollInterval = '30 sec'
}
```

When using two (or more) different executors in your pipeline, you can specify their settings separately by prefixing the executor name with the symbol `$` and using it as special scope identifier. For example:

```
executor {
  $sge {
    queueSize = 100
    pollInterval = '30sec'
  }

  $local {
    cpus = 8
    memory = '32 GB'
  }
}
```

The above configuration example can be rewritten using the dot notation as shown below:

```

executor.$sge.queueSize = 100
executor.$sge.pollInterval = '30sec'
executor.$local.cpus = 8
executor.$local.memory = '32 GB'

```

8.2.5 Scope *docker*

The `docker` configuration scope controls how `Docker` containers are executed by Nextflow.

The following settings are available:

| Name | Description |
|----------------------------|---|
| <code>enabled</code> | Turn this flag to <code>true</code> to enable Docker execution (default: <code>false</code>). |
| <code>legacy</code> | Uses command line options removed since version 1.10.x (default: <code>false</code>). |
| <code>sudo</code> | Executes Docker run command as <code>sudo</code> (default: <code>false</code>). |
| <code>tty</code> | Allocates a pseudo-tty (default: <code>false</code>). |
| <code>temp</code> | Mounts a path of your choice as the <code>/tmp</code> directory in the container. Use the special value <code>auto</code> to create a temporary directory each time a container is created. |
| <code>remove</code> | Clean-up the container after the execution (default: <code>true</code>). For details see: http://docs.docker.com/reference/run/#clean-up-rm . |
| <code>runOptions</code> | This attribute can be used to provide any extra command line options supported by the <code>docker run</code> command. For details see: http://docs.docker.com/reference/run . |
| <code>registry</code> | The registry from where Docker images are pulled. It should be only used to specify a private registry server. It should NOT include the protocol prefix i.e. <code>http://</code> . |
| <code>fixOwnership</code> | Fixes ownership of files created by the docker container. |
| <code>engineOptions</code> | This attribute can be used to provide any option supported by the Docker engine i.e. <code>docker [OPTIONS]</code> . |
| <code>mountFlags</code> | Add the specified flags to the volume mounts e.g. <code>mountFlags = 'ro,Z'</code> |

The above options can be used by prefixing them with the `docker` scope or surrounding them by curly brackets, as shown below:

```

process.container = 'nextflow/examples'

docker {
    enabled = true
    temp = 'auto'
}

```

Read [Docker containers](#) page to learn more how use Docker containers with Nextflow.

8.2.6 Scope *singularity*

The `singularity` configuration scope controls how `Singularity` containers are executed by Nextflow.

The following settings are available:

| Name | Description |
|---------------|--|
| enabled | Turn this flag to <code>true</code> to enable Singularity execution (default: <code>false</code>). |
| engineOptions | This attribute can be used to provide any option supported by the Singularity engine i.e. <code>singularity [OPTIONS]</code> . |
| runOptions | This attribute can be used to provide any extra command line options supported by the <code>singularity exec</code> . |
| autoMounts | When <code>true</code> Nextflow automatically mounts host paths in the executed contained. It requires the <i>user bind control</i> feature enabled in your Singularity installation (default: <code>false</code>). |
| cacheDir | The directory where remote Singularity images are stored. When using a computing cluster it must be a shared folder accessible to all computing nodes. |
| pullTimeout | The amount of time the Singularity pull can last, exceeding which the process is terminated (default: 20 min). |

Read [Singularity containers](#) page to learn more how use Singularity containers with Nextflow.

8.2.7 Scope *manifest*

The `manifest` configuration scope allows you to define some meta-data information needed when publishing your pipeline project on GitHub, BitBucket or GitLab.

The following settings are available:

| Name | Description |
|---------------|---|
| author | Project author name (use a comma to separate multiple names). |
| homePage | Project home page URL |
| description | Free text describing the pipeline project |
| mainScript | Pipeline main script (default: <code>main.nf</code>) |
| defaultBranch | Git repository default branch (default: <code>master</code>) |

The above options can be used by prefixing them with the `manifest` scope or surrounding them by curly brackets. For example:

```
manifest {
  homePage = 'http://foo.com'
  description = 'Pipeline does this and that'
  mainScript = 'foo.nf'
}
```

To learn how to publish your pipeline on GitHub, BitBucket or GitLab code repositories read [Pipeline sharing](#) documentation page.

8.2.8 Scope *trace*

The `trace` scope allows you to control the layout of the execution trace file generated by Nextflow.

The following settings are available:

| Name | Description |
|----------|--|
| en-abled | When <code>true</code> turns on the generation of the execution trace report file (default: <code>false</code>). |
| fields | Comma separated list of fields to be included in the report. The available fields are listed at this page |
| file | Trace file name (default: <code>trace.txt</code>). |
| sep | Character used to separate values in each row (default: <code>\t</code>). |
| raw | When <code>true</code> turns on raw number report generation i.e. date and time are reported as milliseconds and memory as number of bytes |

The above options can be used by prefixing them with the `trace` scope or surrounding them by curly brackets. For example:

```
trace {
  enabled = true
  file = 'pipeline_trace.txt'
  fields = 'task_id,name,status,exit,realtime,%cpu,rss'
}
```

To learn more about the execution report that can be generated by Nextflow read [Trace report](#) documentation page.

8.2.9 Scope *aws*

The `aws` scope allows you to configure the access to Amazon S3 storage. Use the attributes `accessKey` and `secretKey` to specify your bucket credentials. For example:

```
aws {
  accessKey = '<YOUR S3 ACCESS KEY>'
  secretKey = '<YOUR S3 SECRET KEY>'
  region = '<REGION IDENTIFIER>'
}
```

Click the following link to learn more about [AWS Security Credentials](#).

Advanced client configuration options can be set by using the `client` attribute. The following properties can be used:

| Name | Description |
|----------------------------|--|
| connectionTime-out | The amount of time to wait (in milliseconds) when initially establishing a connection before giving up and timing out. |
| endpoint | The AWS S3 API entry point e.g. <i>s3-us-west-1.amazonaws.com</i> . |
| maxConnections | The maximum number of allowed open HTTP connections. |
| maxErrorRetry | The maximum number of retry attempts for failed retryable requests. |
| protocol | The protocol (i.e. HTTP or HTTPS) to use when connecting to AWS. |
| proxyHost | The proxy host to connect through. |
| proxyPort | The port on the proxy host to connect through. |
| proxyUsername | The user name to use when connecting through a proxy. |
| proxyPassword | The password to use when connecting through a proxy. |
| signerOverride | The name of the signature algorithm to use for signing requests made by the client. |
| socketSend-Buffer-SizeHint | The Size hint (in bytes) for the low level TCP send buffer. |
| socketRecvBuffer-SizeHint | The Size hint (in bytes) for the low level TCP receive buffer. |
| socketTimeout | The amount of time to wait (in milliseconds) for data to be transferred over an established, open connection before the connection is timed out. |
| storageEncryption | The S3 server side encryption to be used when saving objects on S3 (currently only AES256 is supported) |
| userAgent | The HTTP user agent header passed with all HTTP requests. |
| uploadMax-Threads | The maximum number of threads used for multipart upload. |
| uploadChunk-Size | The size of a single part in a multipart upload (default: <i>10 MB</i>). |
| uploadStorage-Class | The S3 storage class applied to stored objects, either <i>STANDARD</i> or <i>REDUCED_REDUNDANCY</i> (default: <i>STANDARD</i>). |
| uploadMaxAttempts | The maximum number of upload attempts after which a multipart upload returns an error (default: 5). |
| upload-RetrySleep | The time to wait after a failed upload attempt to retry the part upload (default: <i>100ms</i>). |

For example:

```
aws {
  client {
    maxConnections = 20
    connectionTimeout = 10000
    uploadStorageClass = 'REDUCED_REDUNDANCY'
    storageEncryption = 'AES256'
  }
}
```

8.2.10 Scope *cloud*

The `cloud` scope allows you to define the settings of the computing cluster that can be deployed in the cloud by Nextflow.

The following settings are available:

| Name | Description |
|-----------------------|--|
| bootStorageSize | Boot storage volume size e.g. 10 GB. |
| imageId | Identifier of the virtual machine(s) to launch e.g. ami-43f49030. |
| instanceRole | IAM role granting required permissions and authorizations in the launched instances. When specifying an IAM role no access/security keys are installed in the cluster deployed in the cloud. |
| instanceType | Type of the virtual machine(s) to launch e.g. m4.xlarge. |
| instanceStorageMount | Ephemeral instance storage mount path e.g. /mnt/scratch. |
| instanceStorageDevice | Ephemeral instance storage device name e.g. /dev/xvdc (optional). |
| keyName | SSH access key name given by the cloud provider. |
| keyHash | SSH access public key hash string. |
| keyFile | SSH access public key file path. |
| securityGroup | Identifier of the security group to be applied e.g. sg-df72b9ba. |
| sharedStorageId | Identifier of the shared file system instance e.g. fs-1803efd1. |
| sharedStorageMount | Mount path of the shared file system e.g. /mnt/efs. |
| subnetId | Identifier of the VPC subnet to be applied e.g. subnet-05222a43. |
| spotPrice | Price bid for spot/preemptive instances. |
| userName | SSH access user name (don't specify it to use the image default user name). |
| autoscale | See below. |

The autoscale configuration group provides the following settings:

| Name | Description |
|-------------------|--|
| enabled | Enable cluster auto-scaling. |
| terminateWhenIdle | Enable cluster automatic scale-down i.e. instance terminations when idle (default: <code>false</code>). |
| idleTimeout | Amount of time in idle state after which an instance is candidate to be terminated (default: 5 min). |
| starvingTimeout | Amount of time after which one or more tasks pending for execution trigger an auto-scale request (default: 5 min). |
| minInstances | Minimum number of instances in the cluster. |
| maxInstances | Maximum number of instances in the cluster. |
| imageId | Identifier of the virtual machine(s) to launch when new instances are added to the cluster. |
| instanceType | Type of the virtual machine(s) to launch when new instances are added to the cluster. |
| spotPrice | Price bid for spot/preemptive instances launched while auto-scaling the cluster. |

8.2.11 Scope *conda*

The `conda` scope allows for the definition of the configuration settings that control the creation of a Conda environment by the Conda package manager.

The following settings are available:

| Name | Description |
|----------------|--|
| cacheDir | Defines the path where Conda environments are stored. When using a compute cluster make sure to provide a shared file system path accessible from all computing nodes. |
| create-Timeout | Defines the amount of time the Conda environment creation can last. The creation process is terminated when the timeout is exceeded (default: 20 min). |

8.2.12 Scope *k8s*

The `k8s` scope allows the definition of the configuration settings that control the deployment and execution of workflow applications in a Kubernetes cluster.

The following settings are available:

| Name | Description |
|---------------------|--|
| auto-MountHostPaths | Automatically mounts host paths in the job pods. Only for development purpose when using a single node cluster (default: <code>false</code>). |
| context | Defines the Kubernetes configuration context name to use. |
| namespace | Defines the Kubernetes namespace to use (default: <code>default</code>). |
| serviceAccount | Defines the Kubernetes service account name to use. |
| userDir | Defines the path where the workflow is launched and the user data is stored. This must be a path in a shared K8s persistent volume (default: <code><volume-claim-mount-path>/<user-name></code>). |
| workDir | Defines the path where the workflow temporary data is stored. This must be a path in a shared K8s persistent volume (default: <code><user-dir>/work</code>). |
| projectDir | Defines the path where Nextflow projects are downloaded. This must be a path in a shared K8s persistent volume (default: <code><volume-claim-mount-path>/projects</code>). |
| pod | Allows the definition of one or more pod configuration options such as environment variables, config maps, secrets, etc. It allows the same settings as the pod process directive. |
| volume-Claims | (deprecated) |
| storage-Claim-Name | The name of the persistent volume claim where store workflow result data. |
| storage-Mount-Path | The path location used to mount the persistent volume claim (default: <code>/workspace</code>). |

See the [Kubernetes](#) documentation for more details.

8.2.13 Scope *timeline*

The `timeline` scope allows you to enable/disable the processes execution timeline report generated by Nextflow.

The following settings are available:

| Name | Description |
|---------|--|
| enabled | When <code>true</code> turns on the generation of the timeline report file (default: <code>false</code>). |
| file | Timeline file name (default: <code>timeline.html</code>). |

8.2.14 Scope *mail*

The `mail` scope allows you to define the mail server configuration settings needed to send email messages.

| Name | Description |
|------------------------------|---|
| <code>from</code> | Default email sender address. |
| <code>smtp.host</code> | Host name of the mail server. |
| <code>smtp.port</code> | Port number of the mail server. |
| <code>smtp.user</code> | User name to connect to the mail server. |
| <code>smtp.password</code> | User password to connect to the mail server. |
| <code>smtp.proxy.host</code> | Host name of an HTTP web proxy server that will be used for connections to the mail server. |
| <code>smtp.proxy.port</code> | Port number for the HTTP web proxy server. |
| <code>smtp.*</code> | Any SMTP configuration property supported by the Java Mail API (see link below). |
| <code>debug</code> | When <code>true</code> enables Java Mail logging for debugging purpose. |

Note: Nextflow relies on the [Java Mail API](#) to send email messages. Advanced mail configuration can be provided by using any SMTP configuration property supported by the Java Mail API. See the [table of available properties at this link](#).

For example, the following snippet shows how to configure Nextflow to send emails through the [AWS Simple Email Service](#):

```
mail {
  smtp.host = 'email-smtp.us-east-1.amazonaws.com'
  smtp.port = 587
  smtp.user = '<Your AWS SES access key>'
  smtp.password = '<Your AWS SES secret key>'
  smtp.auth = true
  smtp.starttls.enable = true
  smtp.starttls.required = true
}
```

8.2.15 Scope *notification*

The `notification` scope allows you to define the automatic sending of a notification email message when the workflow execution terminates.

| Name | Description |
|-----------------------|---|
| <code>enabled</code> | Enables the sending of a notification message when the workflow execution completes. |
| <code>to</code> | Recipient address for the notification email. Multiple addresses can be specified separating them with a comma. |
| <code>from</code> | Sender address for the notification email message. |
| <code>template</code> | Path of a template file which provides the content of the notification message. |
| <code>binding</code> | An associative array modelling the variables in the template file. |

The notification message is sent by using the SMTP server defined in the configuration [mail scope](#).

If no mail configuration is provided, it tries to send the notification message by using the external mail command eventually provided by the underlying system (eg. `sendmail` or `mail`).

8.2.16 Scope report

The `report` scope allows you to define configuration setting of the workflow *Execution report*.

| Name | Description |
|----------------------|---|
| <code>enabled</code> | If <code>true</code> it create the workflow execution report. |
| <code>file</code> | The path of the created execution report file (default: <code>report.html</code>). |

8.3 Config profiles

Configuration files can contain the definition of one or more *profiles*. A profile is a set of configuration attributes that can be activated/chosen when launching a pipeline execution by using the `-profile` command line option.

Configuration profiles are defined by using the special scope `profiles` which group the attributes that belong to the same profile using a common prefix. For example:

```
profiles {  
  
  standard {  
    process.executor = 'local'  
  }  
  
  cluster {  
    process.executor = 'sge'  
    process.queue = 'long'  
    process.memory = '10GB'  
  }  
  
  cloud {  
    process.executor = 'cirrus'  
    process.container = 'cbcr/g/imagex'  
    docker.enabled = true  
  }  
  
}
```

This configuration defines three different profiles: `standard`, `cluster` and `cloud` that set different process configuration strategies depending on the target runtime platform. By convention the `standard` profile is implicitly used when no other profile is specified by the user.

Tip: Two or more configuration profiles can be specified by separating the profile names with a comma character, for example:

```
nextflow run <your script> -profile standard,cloud
```

The above feature requires version 0.28.x or higher.

8.4 Environment variables

The following environment variables control the configuration of the Nextflow runtime and the Java virtual machine used by it.

| Name | Description |
|--------------------------|---|
| NXF_HOME | Nextflow home directory (default: <code>\$HOME/.nextflow</code>). |
| NXF_VER | Defines what version of Nextflow to use. |
| NXF_ORG | Default <i>organization</i> prefix when looking for a hosted repository (default: <code>nextflow-io</code>). |
| NXF_GRAB | Provides extra runtime dependencies downloaded from a Maven repository service. |
| NXF_OPTS | Provides extra options for the Java and Nextflow runtime. It must be a blank separated list of <code>-Dkey[=value]</code> properties. |
| NXF_CLASSPATH | Allows the extension of the Java runtime classpath with extra JAR files or class folders. |
| NXF_ASSETS | Defines the directory where downloaded pipeline repositories are stored (default: <code>\$NXF_HOME/assets</code>) |
| NXF_PID_FILE | Name of the file where the process PID is saved when Nextflow is launched in background. |
| NXF_WORK | Directory where working files are stored (usually your <i>scratch</i> directory) |
| NXF_TEMP | Directory where temporary files are stored |
| NXF_DEBUG | Defines scripts debugging level: 1 dump task environment variables in the task log file; 2 enables command script execution tracing; 3 enables command wrapper execution tracing. |
| NXF_EXECUTOR | Defines the default process executor e.g. <i>sge</i> |
| NXF_CONDA_CACHEDIR | Directory where Conda environments are store. When using a computing cluster it must be a shared folder accessible from all computing nodes. |
| NXF_SINGULARITY_CACHEDIR | Directory where remote Singularity images are stored. When using a computing cluster it must be a shared folder accessible from all computing nodes. |
| NXF_JAVA_HOME | Defines the path location of the Java VM installation used to run Nextflow. This variable overrides the <code>JAVA_HOME</code> variable if defined. |
| NXF_OFFLINE | When <code>true</code> disables the project automatic download and update from remote repositories (default: <code>false</code>). |
| JAVA_HOME | Defines the path location of the Java VM installation used to run Nextflow. |
| JAVA_CMD | Defines the path location of the Java binary command used to launch Nextflow. |
| HTTP_PROXY | Defines the HTTP proxy server |
| HTTPS_PROXY | Defines the HTTPS proxy server |

Nextflow provides out of the box support for the Amazon AWS cloud allowing you to setup a computing cluster, deploy it and run your pipeline in the AWS infrastructure in a few commands.

9.1 Configuration

Cloud configuration attributes are provided in the `nextflow.config` file as shown in the example below:

```
cloud {  
    imageId = 'ami-4b7daa32'  
    instanceType = 'm4.xlarge'  
    subnetId = 'subnet-05222a43'  
}
```

The above attributes define the virtual machine ID and type to be used and the VPC subnet ID to be applied in your cluster. Replace these values with the ones of your choice.

Nextflow only requires a Linux image that provides support for [Cloud-init](#) bootstrapping mechanism and includes a Java runtime (version 8) and a Docker engine (version 1.11 or higher).

For your convenience the following pre-configured Amazon Linux AMI is available in the *EU Ireland* region: `ami-4b7daa32`.

9.1.1 AWS credentials

Nextflow will use the AWS credentials defined in your environment, using the standard AWS variables shown below:

- `AWS_ACCESS_KEY_ID`
- `AWS_SECRET_ACCESS_KEY`
- `AWS_DEFAULT_REGION`

Alternatively AWS credentials can be specified in the Nextflow configuration file. See [AWS configuration](#) for more details.

Note: Credentials can also be provided by using an IAM Instance Role. The benefit of this approach is that it spares you from managing/distributing AWS keys explicitly. Read the [IAM Roles](#) documentation and [this blog post](#) for more details.

9.1.2 User & SSH key

By default Nextflow creates in each EC2 instance a user with the same name as the one in your local computer and install the SSH public key available at the path `$HOME/.ssh/id_rsa.pub`. A different user/key can be specified as shown below:

```
cloud {
  userName = 'the-user-name'
  keyFile = '/path/to/ssh/key.pub'
}
```

If you want to use a *key-pair* defined in your AWS account and the default user configured in the AMI, specify the attribute `keyName` in place of `keyFile` and the name of the existing user specifying the `userName` attribute.

9.1.3 Storage

The following storage types can be defined in the cloud instances: *boot*, *instance* and *shared*.

9.1.4 Boot storage

You can set the size of the [root device volume](#) by specifying the attribute `bootStorageSize`. For example:

```
cloud {
  imageId = 'ami-xxx'
  bootStorageSize = '10 GB'
}
```

9.1.5 Instance storage

Amazon instances can provide one or more [ephemeral storage volumes](#), depending the instance type chosen. This storage can be made available by using the `instanceStorageMount` configuration attributes, as shown below:

```
cloud {
  imageId = 'ami-xxx'
  instanceStorageMount = '/mnt/scratch'
}
```

The mount path can be any of your choice.

Note: When the selected instance provides more than one ephemeral storage volume, Nextflow automatically groups all of them together in a single logical volume and mounts it to the specified path. Therefore the resulting instance

storage size is equals to the sum of the sizes of all ephemeral volumes provided by the actual instance (this feature requires Nextflow version 0.27.0 or higher).

If you want to mount a specific instance storage volume, specify the corresponding device name by using the `instanceStorageDevice` setting in the above configuration. See the Amazon documentation for details on [EC2 Instance storage and devices naming](#).

9.1.6 Shared file system

A shared file system can easily be made available in your cloud cluster by using the [Amazon EFS](#) storage. You will only need to specify in the cloud configuration the EFS file system ID and optionally the mount path. For example:

```
cloud {
  imageId = 'ami-xxx'
  sharedStorageId = 'fs-1803efdl'
  sharedStorageMount = '/mnt/shared'
}
```

Note: When the attribute `sharedStorageMount` is omitted the path `/mnt/efs` is used by default.

9.2 Cluster deployment

Once defined the configuration settings in the `nextflow.config` file you can create the cloud cluster by using the following command:

```
nextflow cloud create my-cluster -c <num-of-nodes>
```

The string `my-cluster` identifies the cluster instance. Replace it with a name of your choice.

Finally replace `num-of-nodes` with the actual number of instances that will made-up the cluster. One node is created as *master*, the remaining as *workers*. If the option `-c` is omitted only the *master* node is created.

Warning: You will be charged accordingly the type and the number of instances chosen.

9.3 Pipeline execution

Once the cluster initialization is complete, connect to the *master* node using the SSH command which will be displayed by Nextflow.

Note: On MacOS, use the following command to avoid being asked for a pass-phrase even you haven't defined one:

```
ssh-add -K [private key file]
```

You can run your Nextflow pipeline as usual, the environment is automatically configured to use the *Ignite* executor. If the Amazon EFS storage is specified in the cloud configuration the Nextflow work directory will automatically be set in a shared folder in that file system.

The suggested approach is to run your pipeline downloading it from a public repository such GitHub and to pack the binaries dependencies in a Docker container as described in the [Pipeline sharing](#) section.

9.4 Cluster shutdown

When completed shutdown the cluster instances by using the following command:

```
nextflow cloud shutdown my-cluster
```

9.5 Cluster auto-scaling

Nextflow integration for AWS cloud provides a native support auto-scaling that allows the computing cluster to scale-out or scale-down i.e. add or remove computing nodes dynamically at runtime.

This is a critical feature, especially for pipelines crunching not homogeneous dataset, because it allows the cluster to adapt dynamically to the actual workload computing resources need as they change over the time.

Cluster auto-scaling is enabled by adding the `autoscale` option group in the configuration file as shown below:

```
cloud {
  imageId = 'ami-xxx'
  autoscale {
    enabled = true
    maxInstances = 10
  }
}
```

The above example enables automatic cluster scale-out i.e. new instances are automatically launched and added to the cluster when tasks remain too long in wait status because there aren't enough computing resources available. The `maxInstances` attribute defines the upper limit to which the cluster can grow.

By default unused instances are not removed when are not utilised. If you want to enable automatic cluster scale-down specify the `terminateWhenIdle` attribute in the `autoscale` configuration group.

It is also possible to define a different AMI image ID, type and spot price for instances launched by the Nextflow autoscaler. For example:

```
cloud {
  imageId = 'ami-xxx'
  instanceType = 'm4.large'

  autoscale {
    enabled = true
    spotPrice = 0.15
    minInstances = 5
    maxInstances = 10
    imageId = 'ami-yyy'
    instanceType = 'm4.4xlarge'
    terminateWhenIdle = true
  }
}
```

By doing that it's possible to create a cluster with a single node i.e. the master node. Then the autoscaler will automatically add the missing instances, up to the number defined by the `minInstances` attributes. These will









have a different image and type from the master node and will be launched a *spot instances* because the `spotPrice` attribute has been specified.

9.6 Spot prices

Nextflow includes an handy command to list the current price of EC2 spot instances. Simply type the following command in your shell terminal:

```
nextflow cloud spot-prices
```

It will print the current spot price for all available instances type, similar to the example below:

| TYPE | PRICE | PRICE/CPU | ZONE | DESCRIPTION | CPUS | MEMORY | DISK |
|-------------|--------|-----------|------------|-------------------------|------|----------|---|
| t1.micro | 0.0044 | 0.0044 | eu-west-1c | Linux/UNIX | 1 | 627.7 MB | - |
| m4.4xlarge | 0.1153 | 0.0072 | eu-west-1a | Linux/UNIX (Amazon VPC) | 16 | 64 GB | - |
| m4.10xlarge | 0.2952 | 0.0074 | eu-west-1b | Linux/UNIX (Amazon VPC) | 40 | 160 GB | - |
| m4.large | 0.0155 | 0.0077 | eu-west-1b | Linux/UNIX (Amazon VPC) | 2 | 8 GB | - |
| m4.2xlarge | 0.0612 | 0.0077 | eu-west-1a | Linux/UNIX (Amazon VPC) | 8 | 32 GB | - |
| m4.xlarge | 0.0312 | 0.0078 | eu-west-1a | Linux/UNIX (Amazon VPC) | 4 | 16 GB | - |
| c4.8xlarge | 0.3406 | 0.0095 | eu-west-1c | Linux/UNIX (Amazon VPC) | 36 | 60 GB | - |
| m1.xlarge | 0.0402 | 0.0100 | eu-west-1b | Linux/UNIX | 4 | 15 GB | 4 x 420  |
| ↪GB | | | | | | | |
| c4.4xlarge | 0.1652 | 0.0103 | eu-west-1b | Linux/UNIX (Amazon VPC) | 16 | 30 GB | - |
| c1.xlarge | 0.0825 | 0.0103 | eu-west-1a | Linux/UNIX | 8 | 7 GB | 4 x 420  |
| ↪GB | | | | | | | |
| m1.medium | 0.0104 | 0.0104 | eu-west-1b | Linux/UNIX (Amazon VPC) | 1 | 3.8 GB | 1 x 410  |
| ↪GB | | | | | | | |
| c3.8xlarge | 0.3370 | 0.0105 | eu-west-1a | Linux/UNIX | 32 | 60 GB | 2 x 320  |
| ↪GB | | | | | | | |
| c3.2xlarge | 0.0860 | 0.0108 | eu-west-1c | Linux/UNIX | 8 | 15 GB | 2 x 80  |
| ↪GB | | | | | | | |
| c3.4xlarge | 0.1751 | 0.0109 | eu-west-1c | Linux/UNIX (Amazon VPC) | 16 | 30 GB | 2 x 160  |
| ↪GB | | | | | | | |
| m3.2xlarge | 0.0869 | 0.0109 | eu-west-1c | Linux/UNIX (Amazon VPC) | 8 | 30 GB | 2 x 80  |
| ↪GB | | | | | | | |
| r3.large | 0.0218 | 0.0109 | eu-west-1c | Linux/UNIX | 2 | 15.2 GB | 1 x 32  |
| ↪GB | | | | | | | |
| : | | | | | | | |

It's even possible to refine the showed data by specifying a filtering and ordering criteria. For example:

```
nextflow cloud spot-prices -sort pricecpu -filter "cpus==4"
```

It will only print instance types having 4 cpus and sorting them by the best price per cpu.

9.7 Advanced configuration

Read [Cloud configuration](#) section to learn more about advanced cloud configuration options.

9.8 AWS Batch

Warning: This is an experimental feature and it may change in a future release. It requires Nextflow version 0.26.0 or higher.

AWS Batch is a managed computing service that allows the execution of containerised workloads in the Amazon cloud infrastructure.

Nextflow provides a built-in support for AWS Batch which allows the seamless deployment of a Nextflow pipeline in the cloud offloading the process executions as Batch jobs.

9.8.1 Configuration

- 1 - Make sure your pipeline processes specifies one or more Docker containers by using the *container* directive.
- 2 - Container images need to be published in a Docker registry such as [Docker Hub](#), [Quay](#) or [ECS Container Registry](#) that can be reached by ECS Batch.
- 3 - Specify the AWS Batch *executor* in the pipeline configuration.
- 4 - Specify one or more AWS Batch queues for the execution of your pipeline by using the *queue* directive. Batch queues allow you to bind the execution of a process to a specific computing environment ie. number of CPUs, type of instances (On-demand or Spot), scaling ability, etc. See the [AWS Batch documentation](#) to learn how to setup Batch queues.
- 5 (optional) - Nextflow automatically creates the Batch *Job definitions* needed to execute your pipeline processes. However you may still need to specify a custom *Job Definition* to fine control the configuration settings of a specific job. In this case you can associate a process execution with a *Job definition* of your choice by using the *container* directive and specifying, in place of the Docker image name, the Job definition name prefixed by the `job-definition://` string.

An example `nextflow.config` file is shown below:

```
process.executor = 'awsbatch'
process.queue = 'my-batch-queue'
process.container = 'quay.io/biocontainers/salmon'
aws.region = 'eu-west-1'
```

Note: Nextflow requires to access the AWS command line tool (`aws`) from the container in which the job runs in order to stage the required input files and to copy back the resulting output files in the [S3 storage](#).

The `aws` tool can either be included in container image(s) used by your pipeline execution or installed in a custom AMI that needs to be used in place of the default AMI when configuring the Batch [Computing environment](#).

The latter approach is preferred because it allows the use of existing Docker images without the need to add the AWS CLI tool to them.

Warning: AWS Batch uses the default ECS instance AMI, which has only a 22 GB storage volume which may not be enough for real world data analysis pipelines.

See the section below to learn how to create a custom AWS Batch custom AMI.

9.8.2 Custom AMI

In the EC2 Dashboard, click the *Launch Instance* button, then choose *AWS Marketplace* in the left pane and enter *ECS* in the search box. In result list select *Amazon ECS-Optimized Amazon Linux AMI*, then continue as usual to configure and launch the instance.

Note: The selected instance has a bootstrap volume of 8GB and a second EBS volume 22G for computation which is hardly enough for real world genomic workloads. Make sure to specify an amount of storage in the second volume large enough for the needs of your pipeline execution.

When the instance is running, SSH into it, install the AWS CLI tools as explained below or any other required tool that may be required.

Also make sure the Docker configuration reflects the amount of storage you have specified when launching the instance as shown below:

```
$ docker info | grep -i data
Data file:
Metadata file:
Data Space Used: 500.2 MB
Data Space Total: 1.061 TB
Data Space Available: 1.06 TB
Metadata Space Used: 733.2 kB
Metadata Space Total: 1.074 GB
Metadata Space Available: 1.073 GB
```

The above example shows the Docker data configuration for a 1000GB EBS data volume. See the [ECS Storage documentation](#) for more details.

Warning: The maximum storage size of a single Docker container is by default 10GB, independently the amount of data space available in the underlying volume (see [Base device size](#) for more details).

You can verify the current setting by using this command:

```
$ docker info | grep -i base
Base Device Size: 10.74 GB
```

If your pipeline needs more storage for a single task execution, you will need to specify the `dm.basesize` setting with a proper value in the `/etc/sysconfig/docker-storage` configuration file. See [here](#) and [here](#) for details.

Once done that, create a new AMI by using the *Create Image* option in the EC2 Dashboard or the AWS command line tool.

The new AMI ID needs to be specified when creating the Batch [Computing environment](#).

9.8.3 AWS CLI installation

The AWS cli tool needs to be installed by using a self-contained package manager such as [Conda](#).

The following snippet shows how to install AWS CLI with [Miniconda](#):

```
sudo yum install -y wget
wget https://repo.continuum.io/miniconda/Miniconda3-latest-Linux-x86_64.sh
bash Miniconda3-latest-Linux-x86_64.sh -b -f -p $HOME/miniconda
```

(continues on next page)

(continued from previous page)

```
$HOME/miniconda/bin/conda install -c conda-forge -y awscli  
rm Miniconda3-latest-Linux-x86_64.sh
```

When complete verifies that the AWS cli package works correctly:

```
$ ./miniconda/bin/aws --version  
aws-cli/1.11.120 Python/3.6.3 Linux/4.9.43-17.39.amzn1.x86_64 botocore/1.5.83
```

Note: The `aws` tool will be placed in a directory named `bin` in the main installation folder. Modifying this directory structure, after the installation, will cause the tool to not work properly.

By default Nextflow will assume the AWS CLI tool is directly available in the container. To use an installation from the host image specify the `awscli` parameter in the Nextflow *executor* configuration as shown below:

```
executor.awscli = '/home/ec2-home/miniconda/bin/aws'
```

9.8.4 Pipeline execution

The pipeline can be launched either in a local computer or a EC2 instance. The latter is suggested for heavy or long running workloads.

Pipeline input data should to be stored in the Input data [S3 storage](#). In the same manner the pipeline execution must specifies a S3 bucket as working directory. For example:

```
nextflow run my-pipeline -w s3://my-bucket/some/path
```

9.8.5 Troubleshooting

Problem: The Pipeline execution terminates with an AWS error message similar to the one shown below:

```
JobQueue <your queue> not found
```

Make sure you have defined a AWS region in the Nextflow configuration file and it matches the region in which your Batch environment has been created.

Problem: A process execution fails reporting the following error message:

```
Process <your task> terminated for an unknown reason -- Likely it has been terminated_  
↳by the external system
```

This may happen when Batch is unable to execute the process script. A common cause of this problem is that the Docker container image you have specified uses a non standard [entrypoint](#) which does not allow the execution of the BASH launcher script required by Nextflow to run the job.

Check also the Job execution log in the AWS Batch dashboard for further error details.

CHAPTER 10

Amazon S3 storage

Nextflow includes the support for Amazon S3 storage. Files stored in a S3 bucket can be accessed transparently in your pipeline script like any other file in the local file system.

10.1 S3 path

In order to access a S3 file you only need to prefix the file path with the `s3` schema and the *bucket* name where it is stored.

For example if you need to access the file `/data/sequences.fa` stored in a bucket with name `my-bucket`, that file can be accessed using the following fully qualified path:

```
s3://my-bucket/data/sequences.fa
```

The usual file operations can be applied on a path handle created using the above notation. For example the content of a S3 file can be printed as shown below:

```
println file('s3://my-bucket/data/sequences.fa').text
```

See section *Files and I/O* to learn more about available file operations.

10.2 Security credentials

Amazon access credentials can be provided in two ways:

1. Using AWS access and secret keys in your pipeline configuration.
2. Using IAM roles to grant access to S3 storage on Amazon EC2 instances.

10.2.1 AWS access and secret keys

The AWS access and secret keys can be specified by using the `aws` section in the `nextflow.config` configuration file as shown below:

```
aws {
  accessKey = '<Your AWS access key>'
  secretKey = '<Your AWS secret key>'
  region = '<AWS region identifier>'
}
```

If the access credentials are not found in the above file, Nextflow looks for AWS credentials in a number of different places, including environment variables and local AWS configuration files.

Nextflow looks for AWS credentials in the following order:

1. the `nextflow.config` file in the pipeline execution directory
2. the environment variables `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY`
3. the environment variables `AWS_ACCESS_KEY` and `AWS_SECRET_KEY`
4. the *default* profile in the AWS credentials file located at `~/.aws/credentials`
5. the *default* profile in the AWS client configuration file located at `~/.aws/config`
6. the temporary AWS credentials provided by an IAM instance role. See [IAM Roles](#) documentation for details.

More information regarding [AWS Security Credentials](#) are available in Amazon documentation.

10.2.2 IAM roles Amazon EC2 instances

When running your pipeline into a EC2 instance, IAM roles can be used to grant access to AWS resources.

In this scenario, you only need to launch the EC2 instance specifying a IAM role which includes a *S3 full access* policy. Nextflow will detected and acquire automatically the access grant to the S3 storage, without any further configuration.

Learn more about [Using IAM Roles to Delegate Permissions to Applications that Run on Amazon EC2](#) on Amazon documentation.

10.3 Advanced configuration

Read [AWS configuration](#) section to learn more about advanced S3 client configuration options.

Conda environments

[Conda](#) is an open source package and environment management system that simplifies the installation and the configuration of complex software packages in a platform agnostic manner.

Nextflow has built-in support for Conda that allows the configuration of workflow dependencies using Conda recipes and environment files.

This allows Nextflow applications to use popular tool collections such as [Bioconda](#) whilst taking advantage of the configuration flexibility provided by Nextflow.

11.1 Prerequisites

This feature requires Nextflow version 0.30.x or higher and the Conda or [Miniconda](#) package manager installed on your system.

11.2 How it works

Nextflow automatically creates and activates the Conda environment(s) given the dependencies specified by each process.

Dependencies are specified by using the [conda](#) directive, providing either the names of the required Conda packages, the path of a Conda environment yaml file or the path of an existing Conda environment directory.

Note: Conda environments are stored on the file system. By default Nextflow instructs Conda to save the required environments in the pipeline work directory. Therefore the same environment can be created/saved multiple times across multiple executions when using a different work directory.

You can specify the directory where the Conda environments are stored using the `conda.cacheDir` configuration property (see the [configuration page](#) for details). When using a computing cluster, make sure to use a shared file system path accessible from all computing nodes.

Warning: The Conda environment feature is not supported by executors which use a remote object storage as a work directory eg. AWS Batch.

11.2.1 Use Conda package names

Conda package names can be specified using the `conda` directive. Multiple package names can be specified by separating them with a blank space. For example:

```
process foo {
  conda 'bwa samtools multiqc'

  '''
  your_command --here
  '''
}
```

Using the above definition a Conda environment that includes BWA, Samtools and MultiQC tools is created and activated when the process is executed.

The usual Conda package syntax and naming conventions can be used. The version of a package can be specified after the package name as shown here `bwa=0.7.15`.

The name of the channel where a package is located can be specified prefixing the package with the channel name as shown here `bioconda::bwa=0.7.15`.

11.2.2 Use Conda environment files

Conda environments can also be defined using one or more Conda environment files. This is a file that lists the required packages and channels structured using the YAML format. For example:

```
name: my-env
channels:
  - bioconda
  - conda-forge
  - defaults
dependencies:
  - star=2.5.4a
  - bwa=0.7.15
```

Read the Conda documentation for more details about how to create [environment files](#).

The path of an environment file can be specified using the `conda` directive:

```
process foo {
  conda '/some/path/my-env.yaml'

  '''
  your_command --here
  '''
}
```

Warning: The environment file name **must** end with a `.yaml` or `.yml` suffix otherwise it won't be properly recognised.

Alternatively it is also possible to provide the dependencies using a plain text file, just listing each package name as a separate line. For example:

```
bioconda::star=2.5.4a
bioconda::bwa=0.7.15
bioconda::multiqc=1.4
```

Note: Like before the extension matter, make sure such file ends with the `.txt` extension.

11.2.3 Use existing Conda environments

If you already have a local Conda environment, you can use it in your workflow specifying the installation directory of such environment by using the `conda` directive:

```
process foo {
  conda '/path/to/an/existing/env/directory'

  '''
  your_command --here
  '''
}
```

11.3 Advanced settings

Conda advanced configuration settings are described in the [Conda](#) section on the Nextflow configuration page.

CHAPTER 12

Docker containers

Nextflow integration with [Docker containers](#) technology allows you to write self-contained and truly reproducible computational pipelines.

By using this feature any process in a Nextflow script can be transparently executed into a Docker container. This may be extremely useful to package the binary dependencies of a script into a standard and portable format that can be executed on any platform supporting the Docker engine.

12.1 Prerequisites

You will need Docker installed on your execution environment e.g. your computer or a distributed cluster, depending on where you want to run your pipeline.

If you are running Docker on Mac OSX make sure you are mounting your local `/Users` directory into the Docker VM as explained in this excellent tutorial: [How to use Docker on OSX](#).

12.2 How it works

You won't need to modify your Nextflow script in order to run it with Docker. Simply specify the Docker image from where the containers are started by using the `-with-docker` command line option. For example:

```
nextflow run <your script> -with-docker [docker image]
```

Every time your script launches a process execution, Nextflow will run it into a Docker container created by using the specified image. In practice Nextflow will automatically wrap your processes and run them by executing the `docker run` command with the image you have provided.

Note: A Docker image can contain any tool or piece of software you may need to carry out a process execution. Moreover the container is run in such a way that the process result files are created in the hosting file system, thus it behaves in a completely transparent manner without requiring extra steps or affecting the flow in your pipeline.

If you want to avoid entering the Docker image as a command line parameter, you can define it in the Nextflow configuration file. For example you can add the following lines in the `nextflow.config` file:

```
process.container = 'nextflow/examples:latest'
docker.enabled = true
```

In the above example replace `nextflow/examples:latest` with any Docker image of your choice.

Read the [Configuration](#) page to learn more about the `nextflow.config` file and how to use it to configure your pipeline execution.

Warning: Nextflow automatically manages the file system mounts each time a container is launched depending on the process input files. Note, however, that when a process input is a *symbolic link* file, the linked file **must** be stored in the same folder where the symlink is located, or any its sub-folder. Otherwise the process execution will fail because the launched container won't be able to access the linked file.

12.3 Multiple containers

It is possible to specify a different Docker image for each process definition in your pipeline script. Let's suppose you have two processes named `foo` and `bar`. You can specify two different Docker images for them in the Nextflow script as shown below:

```
process foo {
  container 'image_name_1'

  '''
  do this
  '''
}

process bar {
  container 'image_name_2'

  '''
  do that
  '''
}
```

Alternatively, the same containers definitions can be provided by using the `nextflow.config` file as shown below:

```
process {
  withName:foo {
    container = 'image_name_1'
  }
  withName:bar {
    container = 'image_name_2'
  }
}

docker {
  enabled = true
}
```

Read the [Process scope](#) section to learn more about processes configuration.

12.4 Executable containers

Warning: This feature has been deprecated. It will be removed in a future Nextflow release.

An executable container is a Docker image which defines a command **entry point** i.e. a command that is executed by default when the container starts.

In order to use an executable container with Nextflow set the process' directive `container` to the value `true` and use the Docker image name (including the user/organisation name) as the first command in the process script, followed eventually by any command parameters.

For example:

```
process runImage {
  container true

  '''
  docker/image --foo --bar
  '''
}
```

The docker image name can be preceded by one or more lines containing comments or variables definition i.e. `NAME=VALUE` and can be followed by one or more lines containing BASH commands.

12.5 Advanced settings

Docker advanced configuration settings are described in *Scope docker* section in the Nextflow configuration page.

Singularity containers

[Singularity](#) is a container engine alternative to Docker. The main advantages of Singularity is that it can be used with unprivileged permissions and doesn't require a separate daemon process. These, along other features, like for example the support for autofs mounts, makes Singularity a container engine better suited the requirements of HPC workloads.

Nextflow provides built-in support for Singularity. This allows you to precisely control the execution environment of the processes in your pipeline by running them in isolated containers along all their dependencies.

Moreover the support provided by Nextflow for different container technologies, allows the same pipeline to be transparently executed both with Docker or Singularity containers, depending the available engine in the target execution platforms.

13.1 Prerequisites

You will need Singularity installed on your execution environment e.g. your computer or a distributed cluster, depending on where you want to run your pipeline.

13.2 Images

Singularity makes use of a container image file, which physically contains the container. Refer to the [Singularity documentation](#) to learn how create Singularity images.

Docker images can automatically be converted to Singularity images by using the [docker2singularity](#) tool.

Singularity allows paths that do not currently exist within the container to be created and mounted dynamically by specifying them on the command line. However this feature is only supported on hosts that support the [Overlay file system](#) and is not enabled by default.

Note: Nextflow expects that data paths are defined system wide, and your Singularity images need to be created having the mount paths defined in the container file system.

If your Singularity installation support the *user bind control* feature, enable the Nextflow support for that by defining the `singularity.autoMounts = true` setting in the Nextflow configuration file.

13.3 How it works

The integration for Singularity follows the same execution model implemented for Docker. You won't need to modify your Nextflow script in order to run it with Singularity. Simply specify the Singularity image file from where the containers are started by using the `-with-singularity` command line option. For example:

```
nextflow run <your script> -with-singularity [singularity image file]
```

Every time your script launches a process execution, Nextflow will run it into a Singularity container created by using the specified image. In practice Nextflow will automatically wrap your processes and launch them by running the `singularity exec` command with the image you have provided.

Note: A Singularity image can contain any tool or piece of software you may need to carry out a process execution. Moreover the container is run in such a way that the process result files are created in the hosting file system, thus it behaves in a completely transparent manner without requiring extra steps or affecting the flow in your pipeline.

If you want to avoid entering the Singularity image as a command line parameter, you can define it in the Nextflow configuration file. For example you can add the following lines in the `nextflow.config` file:

```
process.container = '/path/to/singularity.img'
singularity.enabled = true
```

In the above example replace `/path/to/singularity.img` with any Singularity image of your choice.

Read the [Configuration](#) page to learn more about the `nextflow.config` file and how to use it to configure your pipeline execution.

Note: Unlike Docker, Nextflow does not mount automatically host paths in the container when using Singularity. It expects they are configure and mounted system wide by the Singularity runtime. If your Singularity installation allows *user defined bind points* read the [Singularity configuration](#) section to learn how to enable Nextflow auto mounts.

Warning: When a process input is a *symbolic link* file, make sure the linked file **must** is stored in a host folder that is accessible from a bind path defined in your Singularity installation. Otherwise the process execution will fail because the launched container won't be able to access the linked file.

13.4 Multiple containers

It is possible to specify a different Singularity image for each process definition in your pipeline script. For example, let's suppose you have two processes named `foo` and `bar`. You can specify two different Singularity images specifying them in the `nextflow.config` file as shown below:

```
process {
  withName:foo {
    container = 'image_name_1'
  }
}
```

(continues on next page)

(continued from previous page)

```

    withName:bar {
        container = 'image_name_2'
    }
}
singularity {
    enabled = true
}

```

Read the [Process scope](#) section to learn more about processes configuration.

13.5 Singularity & Docker Hub

Nextflow is able to transparently pull remote container images stored in the [Singularity-Hub](#) or any Docker compatible registry.

Note: This feature requires you have installed Singularity 2.3.x or higher

By default when a container name is specified, Nextflow checks if an image file with that name exists in the local file system. If that image file exists, it's used to execute the container. If a matching file does not exist, Nextflow automatically tries to pull an image with the specified name from the Docker Hub.

If you want Nextflow to check only for local file images, prefix the container name with the `file://` pseudo-protocol. For example:

```

process.container = 'file:///path/to/singularity.img'
singularity.enabled = true

```

Warning: Note the use of triple / to specify an **absolute** file path, otherwise the path is interpreted as relative to the workflow launching directory.

To pull images from the Singularity Hub or a third party Docker registry simply prefix the image name with the `shub://` or `docker://` pseudo-protocol as required by the Singularity tool. For example:

```

process.container = 'docker://quay.io/biocontainers/multiqc:1.3--py35_2'
singularity.enabled = true

```

Note: As of Nextflow v0.27 you no longer need to specify `docker://` to pull from a Docker repository. Nextflow will automatically add it to your image name when Singularity is enabled. Additionally, the Docker engine will not work with containers specified as `docker://`.

Note: This feature requires the availability of the `singularity` tool in the computer where the workflow execution is launched (other than the computing nodes).

Nextflow caches those images in the `singularity` directory in the pipeline work directory by default. However it is suggest to provide a centralised caching directory by using either the `NXF_SINGULARITY_CACHEDIR` environment variable or the `singularity.cacheDir` setting in the Nextflow config file.

Warning: When using a computing cluster the Singularity cache directory must be a shared folder accessible to all computing nodes.

Error: When pulling Docker images Singularity may be unable to determine the container size if the image was stored by using an old Docker format, resulting in a pipeline execution error. See the Singularity documentation for details.

13.6 Advanced settings

Singularity advanced configuration settings are described in *Scope singularity* section in the Nextflow configuration page.

CHAPTER 14

Apache Ignite

Nextflow can be deployed in a *cluster* mode by using [Apache Ignite](#), an in-memory data-grid and clustering platform.

Apache Ignite is packaged with Nextflow itself, so you won't need to install it separately or configure other third party software.

14.1 Cluster daemon

In order to setup a cluster you will need to run a cluster daemon on each node within your cluster. If you want to support the [Docker integration](#) feature provided by Nextflow, the Docker engine has to be installed and must run in each node.

In its simplest form just launch the Nextflow daemon in each cluster node as shown below:

```
nextflow node -bg
```

The command line option `-bg` launches the node daemon in the background. The output is stored in the log file `.node-nextflow.log`. The daemon process PID is saved in the file `.nextflow.pid` in the same folder.

14.1.1 Multicast discovery

By default, the Ignite daemon tries to automatically discover all members in the cluster by using the network *multicast* discovery. Note that NOT all networks support this feature (for example Amazon AWS does not).

Tip: To check if multicast is available in your network, use the [iperf](#) tool. To test multicast, open a terminal on two machines within the network and run the following command in the first one `iperf -s -u -B 228.1.2.4 -i 1` and `iperf -c 228.1.2.4 -u -T 32 -t 3 -i 1` on the second. If data is being transferred then multicast is working.

Ignite uses the multicast group `228.1.2.4` and port `47400` by default. You can change these values, by using the `cluster.join` command line option, as shown below:

```
nextflow node -cluster.join multicast:224.2.2.3:55701
```

In case multicast discovery is not available in your network, you can try one of the following alternative methods:

14.1.2 Shared file system

Simply provide a path shared across the cluster by a network file system, as shown below:

```
nextflow node -bg -cluster.join path:/net/shared/cluster
```

The cluster members will use that path to discover each other.

14.1.3 IP addresses

Provide a list of pre-configured IP addresses on the daemon launch command line, for example:

```
nextflow node -cluster.join ip:10.0.2.1,10.0.2.2,10.0.2.4
```

14.1.4 AWS S3 bucket

Creates an Amazon AWS S3 bucket that will hold the cluster member's IP addresses. For example:

```
nextflow node -cluster.join s3:cluster_bucket
```

14.1.5 Advanced options

The following cluster node configuration options can be used.

| Name | Description |
|--------------------|---|
| join | IP address(es) of one or more cluster nodes to which the daemon will join. |
| group | The name of the cluster to which this node join. It allows the creation of separate cluster instances. Default: <code>nextflow</code> |
| maxCpus | Max number of CPUs that can be used by the daemon to run user tasks. By default it is equal to the number of CPU cores. |
| maxDisk | Max amount of disk storage that can be used by user tasks eg. 500 GB. |
| maxMemory | Max amount of memory that can be used by user tasks eg. 64 GB. Default total available memory. |
| interface | Network interfaces that Ignite will use. It can be the interface IP address or name. |
| tcp.localAddress | Defines the local host IP address. |
| tcp.localPort | Defines the local port to listen to. |
| tcp.localPortRange | Range for local ports. |
| tcp.reconnectLimit | Number of times the node tries to (re)establish connection to another node. |
| tcp.networkTimeout | Defines the network timeout. |
| tcp.socketTimeout | Defines the socket operations timeout. This timeout is used to limit connection time and write-to-socket time. Note that when running Ignite on Amazon EC2, socket timeout must be set to a value significantly greater than the default (e.g. to 30000). |
| tcp.ackTimeout | Defines the timeout for receiving acknowledgement for sent message. |
| tcp.maxAckTimeout | Defines the maximum timeout for receiving acknowledgement for sent message. |
| tcp.joinTimeout | Defines the join timeout. |

These options can be specified as command line parameters by adding the prefix `-cluster.` to them, as shown below:

```
nextflow node -bg -cluster.maxCpus 4 -cluster.interface eth0
```

The same options can be entered into the Nextflow *configuration file*, as shown below:

```
cluster {
  join = 'ip:192.168.1.104'
  interface = 'eth0'
}
```

Finally daemon options can be provided also as environment variables having the name in upper-case and by adding the prefix `NXF_CLUSTER_` to them, for example:

```
export NXF_CLUSTER_JOIN='ip:192.168.1.104'
export NXF_CLUSTER_INTERFACE='eth0'
```

14.2 Pipeline execution

The pipeline execution needs to be launched in a *head* node i.e. a cluster node where the Nextflow node daemon is **not** running. In order to execute your pipeline in the Ignite cluster you will need to use the Ignite executor, as shown below:

```
nextflow run <your pipeline> -process.executor ignite
```

If your network does not support multicast discovery, you will need to specify the *joining* strategy as you did for the cluster daemons. For example, using a shared path:

```
nextflow run <your pipeline> -process.executor ignite -cluster.join path:/net/shared/  
↪path
```

14.3 Execution with MPI

Nextflow is able to deploy and self-configure an Ignite cluster on demand, taking advantage of the Open [MPI](#) standard that is commonly available in grid and supercomputer facilities.

In this scenario a Nextflow workflow needs to be executed as an MPI job. Under the hood Nextflow will launch a *driver* process in the first of the nodes, allocated by your job request, and an Ignite daemon in the remaining nodes.

In practice you will need a launcher script to submit an MPI job request to your batch scheduler/resource manager. The batch scheduler must reserve the computing nodes in an exclusive manner to avoid having multiple Ignite daemons running on the same node. Nextflow must be launched using the `mpirun` utility, as if it were an MPI application, specifying the `--pernode` option.

14.3.1 Platform LSF launcher

The following example shows a launcher script for the [Platform LSF](#) resource manager:

```
#!/bin/bash  
#BSUB -oo output_%J.out  
#BSUB -eo output_%J.err  
#BSUB -J <job name>  
#BSUB -q <queue name>  
#BSUB -W 02:00  
#BSUB -x  
#BSUB -n 80  
#BSUB -M 10240  
#BSUB -R "span[ptile=16]"  
export NXF_CLUSTER_SEED=$(shuf -i 0-16777216 -n 1)  
mpirun --pernode nextflow run <your-project-name> -with-mpi [pipeline parameters]
```

It requests 5 nodes (80 processes, with 16 cpus per node). The `-x` directive allocates the node in an exclusive manner. Nextflow needs to be executed using the `-with-mpi` command line option. It will automatically use `ignite` as the executor.

The variable `NXF_CLUSTER_SEED` must contain an integer value (in the range 0-16777216) that will unequivocally identify your cluster instance. In the above example it is randomly generated by using the [shuf](#) Linux tool.

14.3.2 Univa Grid Engine launcher

The following example shows a launcher script for the [Univa Grid Engine](#) (aka SGE):

```
#!/bin/bash  
#$ -cwd  
#$ -j y  
#$ -o <output file name>  
#$ -l virtual_free=10G  
#$ -q <queue name>  
#$ -N <job name>  
#$ -pe omp 5
```

(continues on next page)

(continued from previous page)

```
export NXF_CLUSTER_SEED=$(shuf -i 0-16777216 -n 1)
mpirun --pernode nextflow run <your-project-name> -with-mpi [pipeline parameters]
```

As in the previous script it allocates 5 processing nodes. UGE/SGE does not have an option to reserve a node in an exclusive manner. A common workaround is to request the maximum amount of memory or cpus available in the nodes of your cluster.

14.3.3 Linux SLURM launcher

When using Linux SLURM you will need to use `srun` instead `mpirun` in your launcher script. For example:

```
#!/bin/bash
#SBATCH --job-name=<job name>
#SBATCH --output=<log file %j>
#SBATCH --ntasks=5
#SBATCH --cpus-per-task=16
#SBATCH --tasks-per-node=1
export NXF_CLUSTER_SEED=$(shuf -i 0-16777216 -n 1)
srun nextflow run hello.nf -with-mpi
```

As before, this allocates 5 processing nodes (`--ntasks=5`) and each node will be able to use up to 16 cpus (`--cpus-per-task=16`). When using SLURM it's not necessary to allocate computing nodes in an exclusive manner. It's even possible to launch more than one Nextflow daemon instance per node, though not suggested.

To submit the pipeline execution create a file like the above, then use the following command:

```
sbatch <launcher script name>
```


CHAPTER 15

Kubernetes

Kubernetes is a cloud-native open-source system for deployment, scaling, and management of containerized applications.

It provides clustering and file system abstractions that allows the execution of containerised workloads across different cloud platforms and on-premises installations.

The built-in support for Kubernetes provided by Nextflow streamlines the execution of containerised workflows in Kubernetes clusters.

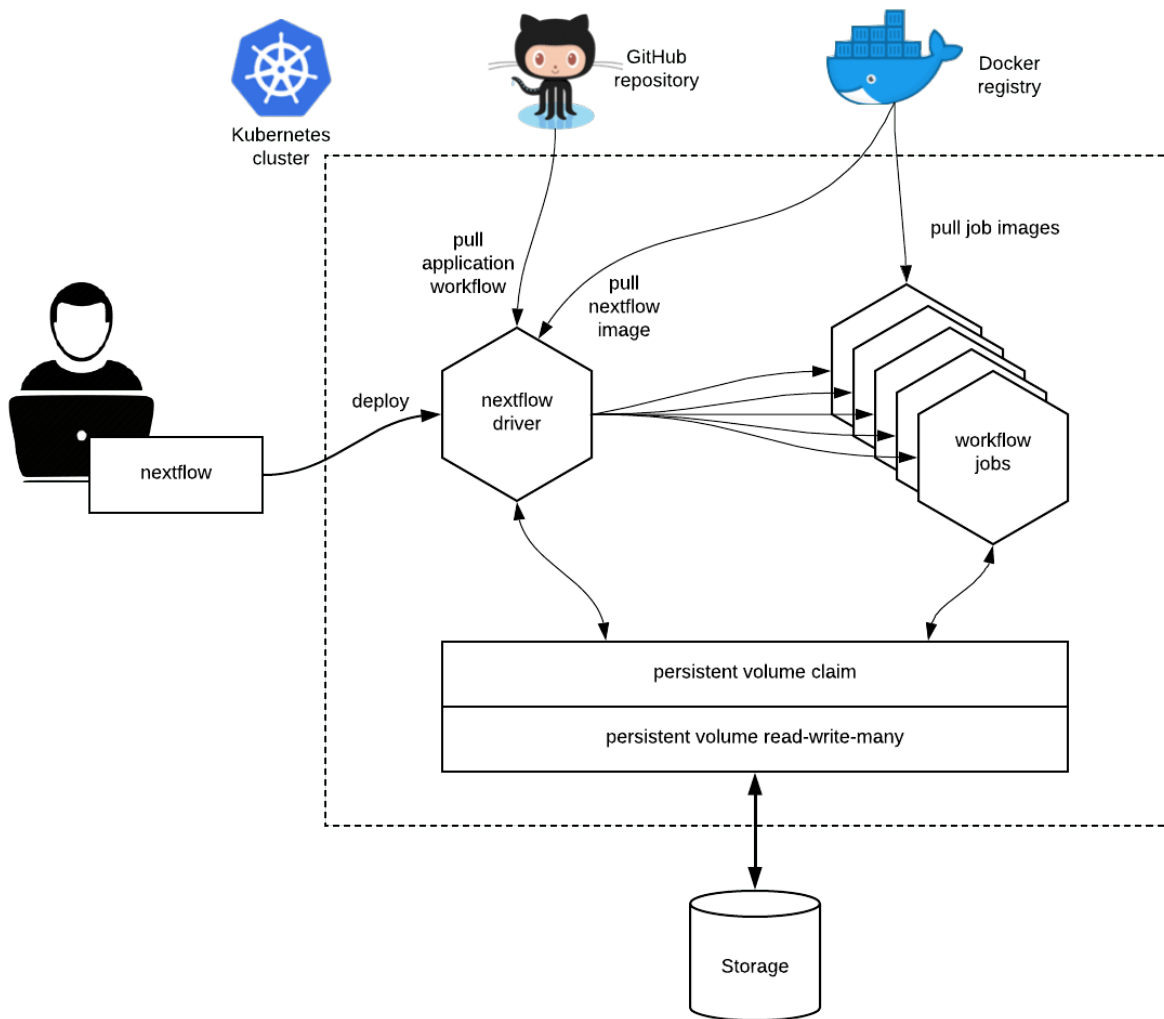
Warning: This is an experimental feature and it may change in a future release. It requires Nextflow version 0.28.0 or higher.

15.1 Concepts

Kubernetes main abstraction is the *pod*. A *pod* defines the (desired) state of one or more containers i.e. required computing resources, storage, network configuration.

Kubernetes abstracts also the storage provisioning through the definition of one more more persistent volumes that allow containers to access to the underlying storage systems in a transparent and portable manner.

When using the `k8s` executor Nextflow deploys the workflow execution as a Kubernetes pod. This pod orchestrates the workflow execution and submits a separate pod execution for each job that need to be carried out by the workflow application.



15.2 Requirements

At least a [Persistent Volume](#) with `ReadWriteMany` access mode has to be defined in the Kubernetes cluster (check the supported storage systems at [this link](#)).

Such volume needs to be accessible through a [Persistent Volume Claim](#), which will be used by Nextflow to run the application and store the scratch data and the pipeline final result.

The workflow application has to be containerised using the usual Nextflow *container* directive.

15.3 Execution

The workflow execution needs to be submitted from a computer able to connect to the Kubernetes cluster.

Nextflow uses the Kubernetes configuration file available at the path `$HOME/.kube/config` or the file specified by the environment variable `KUBECONFIG`.

You can verify such configuration with the command below:

```
$ kubectl cluster-info
Kubernetes master is running at https://your-host:6443
KubeDNS is running at https://your-host:6443/api/v1/namespaces/kube-system/services/
↪ kube-dns:dns/proxy
```

To deploy and launch the workflow execution use the Nextflow command `kuberun` as shown below:

```
nextflow kuberun <pipeline-name> -v vol-claim:/mount/path
```

This command will create and execute a pod running the nextflow orchestrator for the specified workflow. In the above example replace `<pipeline-name>` with an existing nextflow project or the absolute path of a workflow already deployed in the Kubernetes cluster.

The `-v` command line option is required to specify the volume claim name and mount path to use for the workflow execution. In the above example replace `vol-claim` with the name of an existing persistent volume claim and `/mount/path` with the path where the volume is required to be mount in the container. Volume claims can also be specified in the Nextflow configuration file, see the [Kubernetes configuration section](#) for details.

Once the pod execution starts, the application in the foreground prints the console output produced by the running workflow pod.

15.4 Interactive login

For debugging purpose it's possible to execute a Nextflow pod and launch an interactive shell using the following command:

```
nextflow kuberun login -v vol-claim:/mount/path
```

This command creates a pod, sets up the volume claim(s), configures the Nextflow environment and finally launches a Bash login session.

Warning: The pod is automatically destroyed once the shell session terminates. Do not use to start long running workflow executions in background.

15.5 Running in a pod

The main convenience of the `kuberun` command is that it spares the user from manually creating a pod from where the main Nextflow application is launched. In this scenario, the user environment is not containerised.

However there are scenarios in which Nextflow needs to be executed directly from a pod running in a Kubernetes cluster. In these cases you will need to use the plain Nextflow `run` command and specify the `k8s` executor and the required persistent volume claim in the `nextflow.config` file as shown below:

```
process {
    executor = 'k8s'
}

k8s {
    storageClaimName = 'vol-claim'
```

(continues on next page)

(continued from previous page)

```
storageMountPath = '/mount/path'
}
```

In the above snippet replace `vol-claim` with the name of an existing persistent volume claim and replace `/mount/path` with the actual desired mount path eg. `/workspace`.

Warning: The running pod must have been created with the same persistent volume claim name and mount as the one specified in your Nextflow configuration file. Note also that the `run` command does not support the `-v` option.

15.6 Pod settings

The process *pod* directive allows the definition of pods specific settings, such as environment variables, secrets and config maps when using the *Kubernetes* executor. See the *pod* directive for more details.

15.7 Limitation

Currently, the `kuberun` command does not allow the execution of local Nextflow scripts.

15.8 Advanced configuration

Read *Kubernetes configuration* and *executor* sections to learn more about advanced configuration options.

16.1 Execution report

Nextflow can create an HTML execution report: a single document which includes many useful metrics about a workflow execution. The report is organised in the three main sections: *Summary*, *Resources* and *Tasks* (see below for details).


To enable the creation of this report add the `-with-report` command line option when launching the pipeline execution. For example:

```
nextflow run <pipeline name> -with-report [file name]
```

The report file name can be specified as an optional parameter following the report option.

16.1.1 Summary

The *Summary* section reports the execution status, the launch command, overall execution time and some other workflow metadata. You can see an example below:


Nextflow Report
Summary
Resources
Tasks
[angry_babbage]

Nextflow workflow report

[angry_babbage]

Workflow execution completed successfully!

Run times
Sun Nov 05 11:13:07 CET 2017 - Sun Nov 05 13:50:12 CET 2017 (completed 4 days ago, duration: **2h 37m 5s**)

Nextflow command

```
./nextflow-0.26.0-all run main.nf -profile awsbatch --with-report --with-trace -bg --max_samples 38 -w s3://cbcrgeu/work
```

| | |
|--------------------|--|
| Launch directory | /home/pditommaso/projects/rnaseq-encode-nf |
| Work directory | /cbcrgeu/work |
| Project directory | /home/pditommaso/projects/rnaseq-encode-nf |
| Script path | /home/pditommaso/projects/rnaseq-encode-nf/main.nf |
| Script name | main.nf |
| Script hash | 17440c7357d1792c8d6be8223aae92 |
| Workflow container | nextflow/rnaseq-nf |
| Workflow profile | awsbatch |
| Nextflow version | version 0.26.0, build 4710 (03-11-2017 18:14 UTC) |
| Session ID | 54bc9227-daaf-482c-9c62-60ad29af7363 |

16.1.2 Resources

The *Resources* sections plots the distributions of resource usages for each workflow process using the interactive [HighCharts](#) plotting library.

Plots are shown for CPU, memory, time and disk read+write. The first three have two tabs with the raw values and a percentage representation showing what proportion of the allocated resources were used. This is helpful to check that job pipeline requests are efficient.

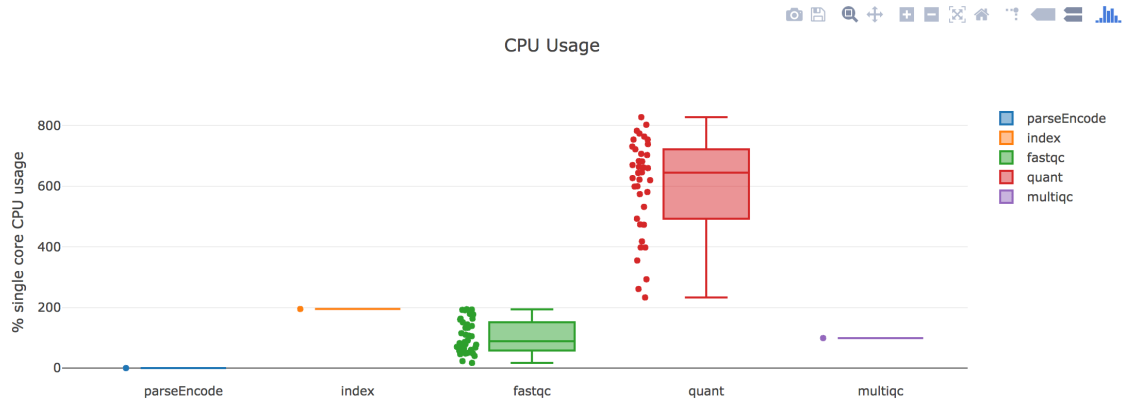
Resource Usage

These plots give an overview of the distribution of resource usage for each process.

CPU Usage

% Allocated

Raw Usage



16.1.3 Tasks

Finally the *Tasks* section lists all executed tasks reporting for each of them, the status, the actual command script and many other runtime metrics. You can see an example below:

Tasks

This table shows information about each task in the workflow. Use the search box on the right to filter rows for specific values. Clicking headers will sort the table by that value and scrolling side to side will reveal more columns.

Show 25 entries

Search:

| task_id | process | tag | status | hash | allocated cpus | %cpu | allocated memory (bytes) | %mem | vmem | rss |
|---------|-------------|---|-----------|-----------|----------------|-------|--------------------------|------|------------|-------|
| 1 | index | Homo_sapiens.GRCh38.cdna.all.fa.i | COMPLETED | f4/a72585 | 2 | 195.0 | 8589934592 | 31.9 | 5272805376 | 51318 |
| 2 | parseEncode | /home/pdittommaso/projects/rnasec encode-nf/data/metadata.tsv | COMPLETED | 12/bdfd13 | 1 | 0.0 | - | 0.0 | 17960960 | 5324 |
| 3 | fastqc | FASTQC on SRR5210435 | COMPLETED | ba/5068a0 | 2 | 46.4 | 6442450944 | 0.0 | 4088819712 | 3685 |
| 4 | fastqc | FASTQC on SRR3192620 | COMPLETED | fa/3e8db3 | 2 | 76.7 | 6442450944 | 0.0 | 4089171968 | 5049 |
| 5 | fastqc | FASTQC on SRR3192621 | FAILED | 6b/f753e2 | 2 | - | 6442450944 | - | - | - |
| 6 | fastqc | FASTQC on SRR3192434 | COMPLETED | 1e/d7f3c2 | 2 | 68.8 | 6442450944 | 0.0 | 4088832000 | 4153 |
| 7 | fastqc | FASTQC on SRR3192433 | COMPLETED | 5e/4886ef | 2 | 70.2 | 6442450944 | 0.0 | 4031012864 | 3843 |

Note: Nextflow collect these metrics running a background process for each job in the target environment. Make sure the following tools are available `ps`, `date`, `sed`, `egrep`, `awk` in the system where the jobs are executed. Moreover some of these metrics are not reported when using a Mac OSX system. See the note message about that in the [Trace report](#) below.

Warning: A common problem when using a third party container image is that it does not ship one or more of the above utilities resulting in an empty execution report.

Please read [Report scope](#) section to learn more about the execution report configuration details.

16.2 Trace report

Nextflow creates an execution tracing file that contains some useful information about each process executed in your pipeline script, including: submission time, start time, completion time, cpu and memory used.

In order to create the execution trace file add the `-with-trace` command line option when launching the pipeline execution. For example:

```
nextflow run <pipeline name> -with-trace
```

It will create a file named `trace.txt` in the current directory. The content looks like the above example:

| task_id | task_name | native_id | name | status | exit | submit | duration | wall-time | %cpu | rss | vmem | rchar | wchar |
|---------|--------------|-----------|----------------|-----------|------|-------------------------|----------|-----------|--------|----------|----------|----------|----------|
| 19 | 45/ab752032 | 2032 | blast (1) | COMPLETED | 0 | 2014-10-23 16:33:16.288 | 1m | 5s | 0.0% | 29.8 MB | 354 MB | 33.3 MB | 0 |
| 20 | 72/db873033 | 2033 | blast (2) | COMPLETED | 0 | 2014-10-23 16:34:17.211 | 30s | 10s | 35.7% | 152.8 MB | 428.1 MB | 192.7 MB | 1 MB |
| 21 | 53/d1318034 | 2034 | blast (3) | COMPLETED | 0 | 2014-10-23 16:34:17.518 | 29s | 20s | 4.5% | 289.5 MB | 381.6 MB | 33.3 MB | 0 |
| 22 | 26/f6512035 | 2035 | blast (4) | COMPLETED | 0 | 2014-10-23 16:34:18.459 | 30s | 9s | 6.0% | 122.8 MB | 353.4 MB | 33.3 MB | 0 |
| 23 | 88/bc002036 | 2036 | blast (5) | COMPLETED | 0 | 2014-10-23 16:34:18.507 | 30s | 19s | 5.0% | 195 MB | 395.8 MB | 65.3 MB | 121 KB |
| 24 | 74/25562037 | 2037 | blast (6) | COMPLETED | 0 | 2014-10-23 16:34:18.553 | 30s | 12s | 43.6% | 140.7 MB | 432.2 MB | 192.7 MB | 182.7 MB |
| 28 | b4/0f962041 | 2041 | exonerate (1) | COMPLETED | 0 | 2014-10-23 16:38:19.657 | 1m 30s | 1m 11s | 94.3% | 611.6 MB | 693.8 MB | 961.2 GB | 6.1 GB |
| 32 | af/7f2f52044 | 2044 | exonerate (4) | COMPLETED | 0 | 2014-10-23 16:46:50.902 | 1m 1s | 38s | 36.6% | 115.8 MB | 167.8 MB | 364 GB | 5.1 GB |
| 33 | 37/ab1f2045 | 2045 | exonerate (5) | COMPLETED | 0 | 2014-10-23 16:47:51.625 | 30s | 12s | 59.6% | 696 MB | 734.6 MB | 354.3 GB | 420.4 MB |
| 31 | d7/eabe2042 | 2042 | exonerate (3) | COMPLETED | 0 | 2014-10-23 16:45:50.846 | 3m 1s | 2m 6s | 130.1% | 703.3 MB | 760.9 MB | 1.1 TB | 28.6 GB |
| 36 | c4/d6cc2048 | 2048 | exonerate (6) | COMPLETED | 0 | 2014-10-23 16:48:48.718 | 3m 1s | 2m 43s | 116.6% | 682.1 MB | 743.6 MB | 868.5 GB | 42 GB |
| 30 | 4f/1ad1f2043 | 2043 | exonerate (2) | COMPLETED | 0 | 2014-10-23 16:45:50.961 | 10m 2s | 9m 16s | 95.5% | 706.2 MB | 764 MB | 1.6 TB | 172.4 GB |
| 52 | 72/41d02055 | 2055 | similarity (1) | COMPLETED | 0 | 2014-10-23 17:13:23.543 | 30s | 352ms | 0.0% | 35.6 MB | 58.3 MB | 199.3 MB | 7.9 MB |
| 57 | 9b/111b2058 | 2058 | similarity (6) | COMPLETED | 0 | 2014-10-23 17:13:23.655 | 30s | 488ms | 0.0% | 108.2 MB | 158 MB | 317.1 MB | 9.8 MB |
| 53 | 3e/bca32061 | 2061 | similarity (2) | COMPLETED | 0 | 2014-10-23 17:13:23.770 | 30s | 238ms | 0.0% | 6.7 MB | 29.6 MB | 190 MB | 91.2 MB |
| 54 | 8b/d45b2062 | 2062 | similarity (3) | COMPLETED | 0 | 2014-10-23 17:13:23.808 | 30s | 442ms | 0.0% | 108.1 MB | 158 MB | 832 MB | 565.6 MB |
| 55 | 51/ac192064 | 2064 | similarity (4) | COMPLETED | 0 | 2014-10-23 17:13:23.873 | 30s | 6s | 0.0% | 112.7 MB | 162.8 MB | 4.9 GB | 3.9 GB |
| 56 | c3/ec5f2066 | 2066 | similarity (5) | COMPLETED | 0 | 2014-10-23 17:13:23.948 | 30s | 616ms | 0.0% | 10.4 MB | 34.6 MB | 238 MB | 8.4 MB |
| 98 | de/d6c02099 | 2099 | matrix (1) | COMPLETED | 0 | 2014-10-23 17:14:27.139 | 30s | 1s | 0.0% | 4.8 MB | 42 MB | 240.6 MB | 79 KB |

The following table shows the fields that can be included in the execution report:

| Name | Description |
|---------------|---|
| task_id | Task ID. |
| hash | Task hash code. |
| native_id | Task ID given by the underlying execution system e.g. POSIX process PID when executed locally, job ID when executed on a cluster. |
| process | Nextflow process name. |
| tag | User provided identifier associated this task. |
| name | Task name. |
| status | Task status. |
| exit | POSIX process exit status. |
| module | Environment module used to run the task. |
| container | Docker image name used to execute the task. |
| cpus | The cpus number request for the task execution. |
| time | The time request for the task execution |
| disk | The disk space request for the task execution. |
| memory | The memory request for the task execution. |
| attempt | Attempt at which the task completed. |
| submit | Timestamp when the task has been submitted. |
| start | Timestamp when the task execution has started. |
| complete | Timestamp when task execution has completed. |
| duration | Time elapsed to complete since the submission. |
| realtime | Task execution time i.e. delta between completion and start timestamp. |
| queue | The queue that the executor attempted to run the process on. |
| %cpu | Percentage of CPU used by the process. |
| %mem | Percentage of memory used by the process. |
| rss | Real memory (resident set) size of the process. Equivalent to <code>ps -o rss</code> . |
| vmem | Virtual memory size of the process. Equivalent to <code>ps -o vsize</code> . |
| * peak_rss | Peak of real memory. This data is read from field <code>VmHWM</code> in <code>/proc/\$pid/status</code> file. |
| * peak_vmem | Peak of virtual memory. This data is read from field <code>VmPeak</code> in <code>/proc/\$pid/status</code> file. |
| * rchar | Number of bytes the process read, using any read-like system call from files, pipes, tty, etc. This data is read from file <code>/proc/\$pid/io</code> . |
| * wchar | Number of bytes the process wrote, using any write-like system call. This data is read from file <code>/proc/\$pid/io</code> . |
| * syscr | Number of read-like system call invocations that the process performed. This data is read from file <code>/proc/\$pid/io</code> . |
| * syscw | Number of write-like system call invocations that the process performed. This data is read from file <code>/proc/\$pid/io</code> . |
| * read_bytes | Number of bytes the process directly read from disk. This data is read from file <code>/proc/\$pid/io</code> . |
| * write_bytes | Number of bytes the process originally dirtied in the page-cache (assuming they will go to disk later). This data is read from file <code>/proc/\$pid/io</code> . |

Note: Fields marked with (*) are not available when running the tracing on Mac OSX. Also note that the Mac OSX default `date` utility, has a time resolution limited to seconds. For a more detailed time tracing it is suggested to install [GNU coreutils](#) package that includes the standard one.

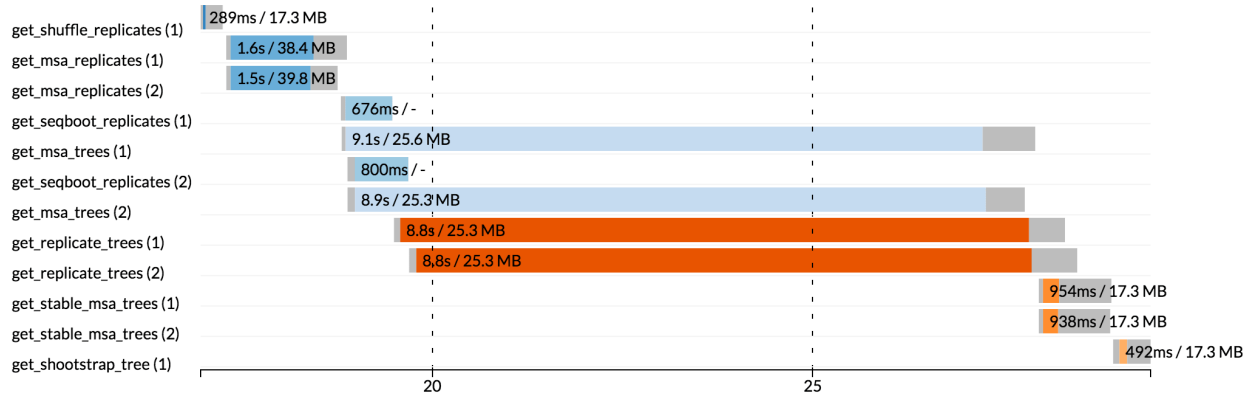
Warning: These numbers provide an estimation of the resources used by running tasks. They should not be intended as an alternative to low level performance analysis provided by other tools and they may not be fully accurate, in particular for very short tasks (taking less than one minute).

Trace report layout and other configuration settings can be specified by using the `nextflow.config` configuration file.

Please read [Trace scope](#) section to learn more about it.

16.3 Timeline report

Nextflow can render an HTML timeline for all processes executed in your pipeline. An example of the timeline report is shown below:



Each bar represents a process run in the pipeline execution. The bar length represents the task duration time (wall-time). The colored area in each bar represents the real execution time. The grey area to the *left* of the colored area represents the task scheduling wait time. The grey area to the *right* of the colored area represents the task termination time (clean-up and file un-staging). The numbers on the x-axis represent the time in absolute units eg. minutes, hours, etc.

Each bar displays two numbers: the task duration time and the virtual memory size peak.

As each process can spawn many tasks, colors are used to identify those tasks belonging to the same process.

To enable the creation of the timeline report add the `-with-timeline` command line option when launching the pipeline execution. For example:

```
nextflow run <pipeline name> -with-timeline [file name]
```

The report file name can be specified as an optional parameter following the timeline option.

16.4 DAG visualisation

A Nextflow pipeline is implicitly modelled by a direct acyclic graph (DAG). The vertices in the graph represent the pipeline's processes and operators, while the edges represent the data connections (i.e. channels) between them.

The pipeline execution DAG can be outputted by adding the `-with-dag` option to the run command line. It creates a file named `dag.dot` containing a textual representation of the pipeline execution graph in the [DOT format](#).

The execution DAG can be rendered in a different format by specifying an output file name which has an extension corresponding to the required format. For example:

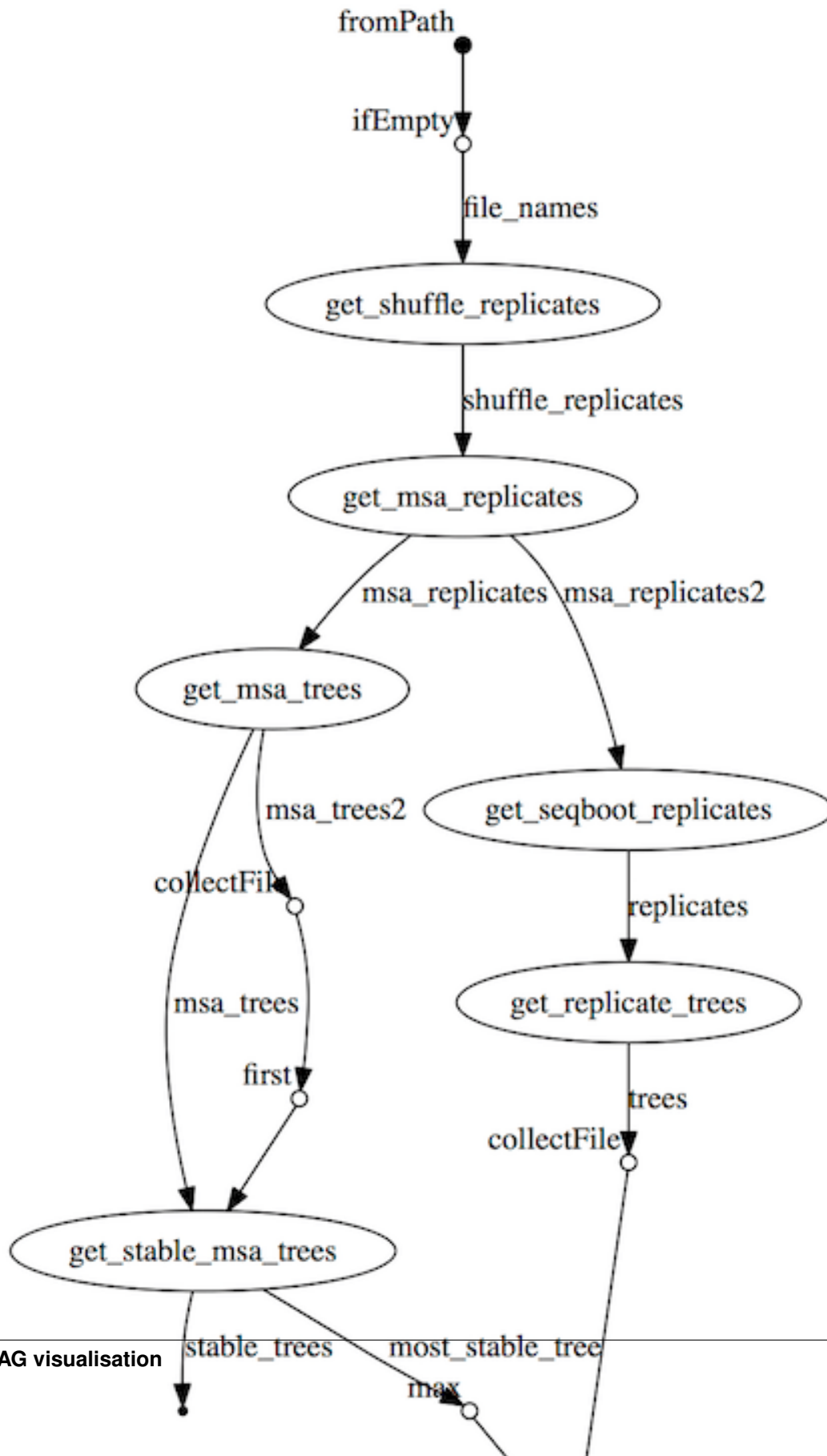
```
nextflow run <script-name> -with-dag flowchart.png
```

List of supported file formats:

| Extension | File format |
|-----------|-------------------|
| dot | Graphviz DOT file |
| html | HTML file |
| pdf | PDF file (*) |
| png | PNG file (*) |
| svg | SVG file (*) |

Warning: The file formats marked with a * require the [Graphviz](#) tool installed in your computer.

The DAG produced by Nextflow for the [Shootstrap](#) pipeline:



Nextflow seamlessly integrates with [BitBucket](#)¹, [GitHub](#), and [GitLab](#) hosted code repositories and sharing platforms. This feature allows you to manage your project code in a more consistent manner or use other people's Nextflow pipelines, published through BitBucket/GitHub/GitLab, in a quick and transparent way.

17.1 How it works

When you launch a script execution with Nextflow, it will look for a file with the pipeline name you've specified. If that file does not exist, it will look for a public repository with the same name on GitHub (unless otherwise specified). If it is found, the repository is automatically downloaded to your computer and executed. This repository is stored in the Nextflow home directory, that is by default the `$HOME/.nextflow` path, and thus will be reused for any further executions.

17.2 Running a pipeline

To launch the execution of a pipeline project, hosted in a remote code repository, you simply need to specify its *qualified* name or the repository URL after the `run` command. The qualified name is formed by two parts: the *owner* name and the *repository* name separated by a `/` character.

In other words if a Nextflow project is hosted, for example, in a GitHub repository at the address `http://github.com/foo/bar`, it can be executed by entering the following command in your shell terminal:

```
nextflow run foo/bar
```

or using the project URL:

```
nextflow run http://github.com/foo/bar
```

¹ BitBucket provides two types of version control system: *Git* and *Mercurial*. Nextflow supports only *Git* based repositories.

Note: In the first case, if your project is hosted on a service other than GitHub, you will need to specify this hosting service in the command line by using the `-hub` option. For example `-hub bitbucket` or `-hub gitlab`. In the second case, i.e. when using the project URL as name, the `-hub` option is not needed.

You can try this feature out by simply entering the following command in your shell terminal:

```
nextflow run nextflow-io/hello
```

It will download a trivial *Hello* example from the repository published at the following address <http://github.com/nextflow-io/hello> and execute it in your computer.

If the *owner* part in the pipeline name is omitted, Nextflow will look for a pipeline between the ones you have already executed having a name that matches the name specified. If none is found it will try to download it using the *organisation* name defined by the environment variable `NXF_ORG` (which by default is `nextflow-io`).

Tip: To access a private repository, specify the access credentials by using the `-user` command line option, then the program will ask you to enter the password interactively. Private repositories access credentials can also be defined in the *SCM configuration file*.

17.3 Handling revisions

Any Git branch, tag or commit ID defined in a project repository, can be used to specify the revision that you want to execute when launching a pipeline by adding the `-r` option to the run command line. So for example you could enter:

```
nextflow run nextflow-io/hello -r mybranch
```

or

```
nextflow run nextflow-io/hello -r v1.1
```

It will execute two different project revisions corresponding to the Git tag/branch having that names.

17.4 Commands to manage projects

The following commands allows you to perform some basic operations that can be used to manage your projects.

Note: Nextflow is not meant to replace functionalities provided by the [Git](#) tool. You may still need it to create new repositories or commit changes, etc.

17.4.1 Listing available projects

The `list` command allows you to list all the projects you have downloaded in your computer. For example:

```
nextflow list
```

This prints a list similar to the following one:

```
cbcrg/ampa-nf
cbcrg/piper-nf
nextflow-io/hello
nextflow-io/examples
```

17.4.2 Showing project information

By using the `info` command you can show information from a downloaded project. For example:

```
project name: nextflow-io/hello
repository   : http://github.com/nextflow-io/hello
local path   : $HOME/.nextflow/assets/nextflow-io/hello
main script  : main.nf
revisions    :
* master (default)
  mybranch
  v1.1 [t]
  v1.2 [t]
```

Starting from the top it shows: 1) the project name; 2) the Git repository URL; 3) the local folder where the project has been downloaded; 4) the script that is executed when launched; 5) the list of available revisions i.e. branches and tags. Tags are marked with a `[t]` on the right, the current checked-out revision is marked with a `*` on the left.

17.4.3 Pulling or updating a project

The `pull` command allows you to download a project from a GitHub repository or to update it if that repository has already been downloaded. For example:

```
nextflow pull nextflow-io/examples
```

Alternatively, you can use the repository URL as the name of the project to pull:

```
nextflow pull https://github.com/nextflow-io/examples
```

Downloaded pipeline projects are stored in the folder `$HOME/.nextflow/assets` in your computer.

17.4.4 Viewing the project code

The `view` command allows you to quickly show the content of the pipeline script you have downloaded. For example:

```
nextflow view nextflow-io/hello
```

By adding the `-l` option to the example above it will list the content of the repository.

17.4.5 Cloning a project into a folder

The `clone` command allows you to copy a Nextflow pipeline project to a directory of your choice. For example:

```
nextflow clone nextflow-io/hello target-dir
```

If the destination directory is omitted the specified project is cloned to a directory with the same name as the pipeline base name (e.g. *hello*) in the current folder.

The clone command can be used to inspect or modify the source code of a pipeline project. You can eventually commit and push back your changes by using the usual Git/GitHub workflow.

17.4.6 Deleting a downloaded project

Downloaded pipelines can be deleted by using the drop command, as shown below:

```
nextflow drop nextflow-io/hello
```

17.5 SCM configuration file

The file `$HOME/.nextflow/scm` allows you to centralise the security credentials required to access private project repositories on Bitbucket, GitHub and GitLab source code management (SCM) platforms or to manage the configuration properties of private server installations (of the same platforms).

The configuration properties for each SCM platform are defined inside the `providers` section, properties for the same provider are grouped together with a common name and delimited with curly brackets as in this example:

```
providers {
  <provider-name> {
    property = value
    :
  }
}
```

In the above template replace `<provider-name>` with one of the “default” servers (i.e. `bitbucket`, `github` or `gitlab`) or a custom identifier representing a private SCM server installation.

The following configuration properties are supported for each provider configuration:

| Name | Description |
|-----------------|--|
| user | User name required to access private repositories on the SCM server. |
| password | User password required to access private repositories on the SCM server. |
| token | Private API access token (used only when the specified platform is <code>gitlab</code>). |
| * plat- form | SCM platform name, either: <code>github</code> , <code>gitlab</code> or <code>bitbucket</code> . |
| * server | SCM server name including the protocol prefix e.g. <code>https://github.com</code> . |
| * end- point | SCM API <i>endpoint</i> URL e.g. <code>https://api.github.com</code> (default: the same value specified for <code>server</code>). |

The attributes marked with a `*` are only required when defining the configuration of a private SCM server.

17.5.1 BitBucket credentials

Create a `bitbucket` entry in the *SCM configuration file* specifying your user name and password, as shown below:

```
providers {

    bitbucket {
        user = 'me'
        password = 'my-secret'
    }

}
```

17.5.2 GitHub credentials

Create a github entry in the *SCM configuration file* specifying your user name and password as shown below:

```
providers {

    github {
        user = 'me'
        password = 'my-secret'
    }

}
```

Tip: You can use use a [Personal API token](#) in place of your GitHub password.

17.5.3 GitLab credentials

Create a gitlab entry in the *SCM configuration file* specifying the user name, password and your API access token that can be found in your GitLab [account page](#) (sign in required). For example:

```
providers {

    gitlab {
        user = 'me'
        password = 'my-secret'
        token = 'YgpR8m7viH_ZYnC8YSe8'
    }

}
```

17.6 Private server configuration

Nextflow is able to access repositories hosted on private BitBucket, GitHub and GitLab server installations.

In order to use a private SCM installation you will need to set the server name and access credentials in your *SCM configuration file*.

If, for example, the host name of your private GitLab server is `gitlab.acme.org`, you will need to have in the `$HOME/.nextflow/scm` file a configuration like the following:

```
providers {  
  
    mygit {  
        server = 'http://gitlab.acme.org'  
        platform = 'gitlab'  
        user = 'your-user'  
        password = 'your-password'  
        token = 'your-api-token'  
    }  
  
}
```

Then you will be able to run/pull a project with Nextflow using the following command line:

```
$ nextflow run foo/bar -hub mygit
```

Or, in alternative, using the Git clone URL:

```
$ nextflow run http://gitlab.acme.org/foo/bar.git
```

Warning: When accessing a private SCM installation over `https` and that server uses a custom SSL certificate you may need to import such certificate into your local Java keystore. Read more [here](#).

17.7 Local repository configuration

Nextflow is also able to handle repositories stored in a local or shared file system. The repository must be created as a [bare repository](#).

Having, for example, a bare repository store at path `/shared/projects/foo.git`, Nextflow is able to run it using the following syntax:

```
$ nextflow run file:/shared/projects/foo.git
```

See [Git documentation](#) for more details about how create and manage bare repositories.

17.8 Publishing your pipeline

In order to publish your Nextflow pipeline to GitHub (or any other supported platform) and allow other people to use it, you only need to create a GitHub repository containing all your project script and data files. If you don't know how to do it, follow this simple tutorial that explains how [create a GitHub repository](#).

Nextflow only requires that the main script in your pipeline project is called `main.nf`. A different name can be used by specifying the `manifest.mainScript` attribute in the `nextflow.config` file that must be included in your project. For example:

```
manifest.mainScript = 'my_very_long_script_name.nf'
```

To learn more about this and other project metadata information, that can be defined in the Nextflow configuration file, read the [Manifest](#) section on the Nextflow configuration page.

Once you have uploaded your pipeline project to GitHub other people can execute it simply using the project name or the repository URL.

For if your GitHub account name is `foo` and you have uploaded a project into a repository named `bar` the repository URL will be `http://github.com/foo/bar` and people will be able to download and run it by using either the command:

```
nextflow run foo/bar
```

or

```
nextflow run http://github.com/foo/bar
```

See the [Running a pipeline](#) section for more details on how to run Nextflow projects.

17.9 Manage dependencies

Computational pipelines are rarely composed by a single script. In real world applications they depend on dozens of other components. These can be other scripts, databases, or applications compiled for a platform native binary format.

External dependencies are the most common source of problems when sharing a piece of software, because the users need to have an identical set of tools and the same configuration to be able to use it. In many cases this has proven to be a painful and error prone process, that can severely limit the ability to reproduce computational results on a system other than the one on which it was originally developed.

Nextflow tackles this problem by integrating GitHub, BitBucket and GitLab sharing platforms and [Docker](#) containers technology.

The use of a code management system is important to keep together all the dependencies of your pipeline project and allows you to track the changes of the source code in a consistent manner.

Moreover to guarantee that a pipeline is reproducible it should be self-contained i.e. it should have ideally no dependencies on the hosting environment. By using Nextflow you can achieve this goal following these methods:

17.9.1 Third party scripts

Any third party script that does not need to be compiled (BASH, Python, Perl, etc) can be included in the pipeline project repository, so that they are distributed with it.

Grant the execute permission to these files and copy them into a folder named `bin/` in the root directory of your project repository. Nextflow will automatically add this folder to the `PATH` environment variable, and the scripts will automatically be accessible in your pipeline without the need to specify an absolute path to invoke them.

17.9.2 System environment

Any environment variable that may be required by the tools in your pipeline can be defined in the `nextflow.config` file by using the `env` scope and including it in the root directory of your project. For example:

```
env {  
    DELTA = 'foo'  
    GAMMA = 'bar'  
}
```

See the [Configuration](#) page to learn more about the Nextflow configuration file.

17.9.3 Resource manager

When using Nextflow you don't need to write the code to parallelize your pipeline for a specific grid engine/resource manager because the parallelization is defined implicitly and managed by the Nextflow runtime. The target execution environment is parametrized and defined in the configuration file, thus your code is free from this kind of dependency.

17.9.4 Bootstrap data

Whenever your pipeline requires some files or dataset to carry out any initialization step, you can include this data in the pipeline repository itself and distribute them together.

To reference this data in your pipeline script in a portable manner (i.e. without the need to use a static absolute path) use the implicit variable `baseDir` which locates the base directory of your pipeline project.

For example, you can create a folder named `dataset/` in your repository root directory and copy there the required data file(s) you may need, then you can access this data in your script by writing:

```
sequences = file("$baseDir/dataset/sequences.fa")
sequences.splitFasta {
    println it
}
```

17.9.5 User inputs

Nextflow scripts can be easily parametrised to allow users to provide their own input data. Simply declare on the top of your script all the parameters it may require as shown below:

```
params.my_input = 'default input file'
params.my_output = 'default output path'
params.my_flag = false
..
```

The actual parameter values can be provided when launching the script execution on the command line by prefixed the parameter name with a double minus character i.e. `--`, for example:

```
nextflow run <your pipeline> --my_input /path/to/input/file --my_output /other/path --
↪my_flag true
```

17.9.6 Binary applications

Docker allows you to ship any binary dependencies that you may have in your pipeline to a portable image that is downloaded on-demand and can be executed on any platform where a Docker engine is installed.

In order to use it with Nextflow, create a Docker image containing the tools needed by your pipeline and make it available in the [Docker registry](#).

Then declare in the `nextflow.config` file, that you will include in your project, the name of the Docker image you have created. For example:

```
process.container = 'my-docker-image'
docker.enabled = true
```

In this way when you launch the pipeline execution, the Docker image will be automatically downloaded and used to run your tasks.

Read the [Docker containers](#) page to learn more on how to use Docker containers with Nextflow.

This mix of technologies makes it possible to write self-contained and truly reproducible pipelines which require zero configuration and can be reproduced in any system having a Java VM and a Docker engine installed.

Workflow introspection

18.1 Runtime metadata

The implicit `workflow` object allows you to access some workflow and runtime metadata in your Nextflow scripts. For example:

```
println "Project : $workflow.projectDir"
println "Git info: $workflow.repository - $workflow.revision [$workflow.commitId]"
println "Cmd line: $workflow.commandLine"
```

Tip: To shortcut the access to multiple `workflow` properties you can use the Groovy `with` method.

The following table lists the properties that can be accessed on the `workflow` object:

| Name | Description |
|-----------------|--|
| scriptId | Project main script unique hash ID. |
| script-Name | Project main script file name. |
| scriptFile | Project main script file path. |
| repository | Project repository Git remote URL. |
| commitId | Git commit ID of the executed workflow repository. |
| revision | Git branch/tag of the executed workflow repository. |
| projectDir | Directory where the workflow project is stored in the computer. |
| launchDir | Directory where the workflow execution has been launched. |
| workDir | Workflow working directory. |
| config-Files | Configuration files used for the workflow execution. |
| container | Docker image used to run workflow tasks. When more than one image is used it returns a map object containing <i>[process name, image name]</i> pair entries. |
| containerEngine | Returns the name of the container engine (e.g. docker or singularity) or null if no container engine is enabled. |
| command-Line | Command line as entered by the user to launch the workflow execution. |
| profile | Used configuration profile. |
| runName | Mnemonic name assigned to this execution instance. |
| sessionId | Unique identifier (UUID) associated to current execution. |
| resume | Returns <code>true</code> whenever the current instance is resumed from a previous execution. |
| start | Timestamp of workflow at execution start. |
| complete | Timestamp of workflow when execution is completed. |
| duration | Time elapsed to complete workflow execution. |
| * success | Reports if the execution completed successfully. |
| * exitStatus | Exit status of the task that caused the workflow execution to fail. |
| * errorMessage | Error message of the task that caused the workflow execution to fail. |
| * errorReport | Detailed error of the task that caused the workflow execution to fail. |

Properties marked with a `a` are accessible only in the workflow completion handler.

Properties marked with a `*` are accessible only in the workflow completion and error handlers. See the [Completion handler](#) section for details.

18.2 Nextflow metadata

The implicit `nextflow` object allows you to access the metadata information of the Nextflow runtime.

| Name | Description |
|--------------------|-------------------------------------|
| nextflow.version | Nextflow runtime version number. |
| nextflow.build | Nextflow runtime build number. |
| nextflow.timestamp | Nextflow runtime compile timestamp. |

The method `nextflow.version.matches` allows you to check if the Nextflow runtime satisfies the version requirement eventually needed by your workflow script. The required version string can be prefixed with the usual comparison operators eg `>`, `>=`, `=`, etc. or postfixed with the `+` operator to specify a minimal version requirement. For example:

```
if( !nextflow.version.matches('0.22+') ) {
    println "This workflow requires Nextflow version 0.22 or greater -- You are_
↪running version $nextflow.version"
    exit 1
}
```

18.3 Completion handler

Due to the asynchronous nature of Nextflow the termination of a script does not correspond to the termination of the running workflow. Thus some information, only available on execution completion, needs to be accessed by using an asynchronous handler.

The `onComplete` event handler is invoked by the framework when the workflow execution is completed. It allows one to access the workflow termination status and other useful information. For example:

```
workflow.onComplete {
    println "Pipeline completed at: $workflow.complete"
    println "Execution status: ${ workflow.success ? 'OK' : 'failed' }"
}
```

18.4 Error handler

The `onError` event handler is invoked by Nextflow when a runtime or process error caused the pipeline execution to stop. For example:

```
workflow.onError {
    println "Oops... Pipeline execution stopped with the following message: $
↪{workflow.errorMessage}"
}
```

Note: Both the `onError` and `onComplete` handlers are invoked when an error condition is encountered. However the first is called as soon as the error is raised, while the second just before the pipeline execution is going terminate. When using the `finish errorStrategy`, between the two there could be a significant time gap depending by the time required to complete any pending job.

18.5 Notification message

Nextflow does not provide a built-in mechanism to send emails or other messages. However the `mail` standard Linux tool (or an equivalent one) can easily be used to send a notification message when the workflow execution is completed, as shown below:

```
workflow.onComplete {
  def subject = 'My pipeline execution'
  def recipient = 'me@gmail.com'

  ['mail', '-s', subject, recipient].execute() << """

  Pipeline execution summary
  -----
  Completed at: ${workflow.complete}
  Duration      : ${workflow.duration}
  Success       : ${workflow.success}
  workDir       : ${workflow.workDir}
  exit status   : ${workflow.exitStatus}
  Error report:  ${workflow.errorReport ?: '-'}
  """
}
```

18.6 Decoupling metadata

The workflow event handlers can be defined also in the `nextflow.config` file. This is useful to decouple the handling of pipeline events from the main script logic.

When the event handlers are included in a configuration file the only difference is that the `onComplete` and the `onError` closures have to be defined by using the assignment operator as shown below:

```
workflow.onComplete = {
  // any workflow property can be used here
  println "Pipeline complete"
  println "Command line: $workflow.commandLine"
}

workflow.onError = {
  println "Oops .. something when wrong"
}
```

Note: It is possible to define a workflow event handlers both in the pipeline script **and** in the configuration file.

19.1 Mail message

The built-in function `sendMail` allows you to send a mail message from a workflow script.

19.1.1 Basic mail

The mail attributes are specified as named parameters or providing an equivalent associative array as argument. For example:

```
sendMail( to: 'you@gmail.com',  
          subject: 'Catch up',  
          body: 'Hi, how are you!',  
          attach: '/some/path/attachment/file.txt' )
```

therefore this is equivalent to write:

```
mail = [ to: 'you@gmail.com',  
          subject: 'Catch up',  
          body: 'Hi, how are you!',  
          attach: '/some/path/attachment/file.txt' ]  
  
sendMail(mail)
```

The following parameters can be specified:

| Name | Description |
|---------|--|
| to * | The mail target recipients. |
| cc * | The mail CC recipients. |
| bcc * | The mail BCC recipients. |
| from * | The mail sender address. |
| subject | The mail subject. |
| charset | The mail content charset (default: UTF-8). |
| text | The mail plain text content. |
| body | The mail body content. It can be either plain text or HTML content. |
| type | The mail body mime type. If not specified it's automatically detected. |
| attach | Single file or a list of files to be included as mail attachments. |

* Multiple email addresses can be specified separating them with a comma.

19.1.2 Advanced mail

An second version of the `sendMail` allows a more idiomatic syntax:

```
sendMail {
  to 'you@gmail.com'
  from 'me@gmail.com'
  attach '/some/path/attachment/file.txt'
  attach '/other/path/image.png'
  subject 'Catch up'

  '''
  Hi there,
  Look! Multi-lines
  mail content!
  '''
}
```

The same attributes listed in the table in the previous section are allowed.

Note: When it terminates with a string expression it's implicitly interpreted as the mail body content, therefore the `body` parameter can be omitted as shown above.

Tip: To send an *alternative* mail message that includes either text and HTML content use both the `text` and `body` attributes. The first must be used for the plain text content, while the second for the rich HTML message.

19.1.3 Mail attachments

When using the curly brackets syntax, the `attach` parameter can be repeated two or more times to include multiple attachments in the mail message.

Moreover for each attachment it's possible to specify one or more of the following optional attributes:

| Name | Description |
|-------------|--|
| contentId | Defines the <i>Content-ID</i> header field for the attachment. |
| disposition | Defines the <i>Content-Disposition</i> header field for the attachment. |
| fileName | Defines the <i>filename</i> parameter of the “Content-Disposition” header field. |
| description | Defines the <i>Content-Description</i> header field for the attachment. |

For example:

```
sendMail {
  to 'you@dot.com'
  attach '/some/file.txt', fileName: 'manuscript.txt'
  attach '/other/image.png', disposition: 'inline'
  subject 'Sending documents'
  '''
  the mail body
  '''
}
```

19.1.4 Mail configuration

If no mail server configuration is provided, Nextflow tries to send the email by using the external mail command eventually provided by the underlying system (eg. sendmail or mail).

If your system does not provide access to none of the above you can configure a SMTP server in the `nextflow.config` file. For example:

```
mail {
  smtp.host = 'your.smtp-server.com'
  smtp.port = 475
  smtp.user = 'my-user'
}
```

See the [mail scope](#) section to learn more the mail server configuration options.

19.2 Mail notification

You can use the `sendMail` function with a *workflow completion handler* to notify the completion of a workflow completion. For example:

```
workflow.onComplete {

  def msg = """\
  Pipeline execution summary
  -----
  Completed at: ${workflow.complete}
  Duration    : ${workflow.duration}
  Success     : ${workflow.success}
  workDir     : ${workflow.workDir}
  exit status : ${workflow.exitStatus}
  """
  .stripIndent()
}
```

(continues on next page)

(continued from previous page)

```
sendMail(to: 'you@gmail.com', subject: 'My pipeline execution', body: msg)
}
```

This is useful to send a custom notification message. Note however that Nextflow includes a built-in notification mechanism which is the most convenient way to notify the completion of a workflow execution in most cases. Read the following section to learn about it.

19.3 Workflow notification

Nextflow includes a built-in workflow notification features that automatically sends a notification message when a workflow execution terminates.

To enable simply specify the `-N` option when launching the pipeline execution. For example:

```
nextflow run <pipeline name> -N <recipient address>
```

It will send a notification mail when the execution completes similar to the one shown below:

Workflow completion notification

Run Name: modest_jones

Execution completed successfully!

The command used to launch the workflow was as follows:

```
./launch.sh run hello -N paolo.ditommaso@gmail.com
```

Execution summary

| | |
|--------------------|---|
| Launch time | 26-Dec-2017 22:35:23 |
| Ending time | 26-Dec-2017 22:35:23 (duration: 494ms) |
| Total CPU-Hours | (a few seconds) |
| Launch directory | /Users/pditommaso/projects/nextflow |
| Work directory | /Users/pditommaso/projects/nextflow/work |
| Project directory | /Users/pditommaso/.nextflow/assets/nextflow-io/hello |
| Script name | main.nf |
| Script ID | 5b832a0bafec15c2db86c5e7fac1e505 |
| Workflow session | 569ac042-258b-41b0-a1db-346625aa5438 |
| Workflow repo | https://github.com/nextflow-io/hello.git |
| Workflow revision | master (1a373400ca072ac953f52d0b73b5a4d1fdeda2d3) |
| Workflow profile | standard |
| Workflow container | - |
| Container engine | - |
| Nextflow version | 0.27.0-RC1, build 4757 (25-12-2017 20:48 UTC) |

This email was sent by Nextflow
cite [doi:10.1038/nbt.3820](https://doi.org/10.1038/nbt.3820)
<http://nextflow.io>

nextflow

Warning: By default the notification message is sent by using the `sendmail` system tool which is assumed to be available in the computer where Nextflow is running. Make sure it's properly installed and configured. Alternatively provide the SMTP server configuration settings to use the Nextflow built-in mail support, which doesn't require any external system tool.

See the [Mail configuration](#) section to learn about the available mail delivery options and configuration settings.

Read [Notification scope](#) section to learn more about the workflow notification configuration details.

20.1 Basic pipeline

This example shows a pipeline that is made of two processes. The first process receives a **FASTA formatted** file and splits it into file chunks whose names start with the prefix `seq_`.

The process that follows, receives these files and it simply *reverses* their content by using the `rev` command line tool.

In more detail:

- line 1: The script starts with a **shebang** declaration. This allows you to launch your pipeline, as any other BASH script
- line 3: Declares a pipeline parameter named `params.in` that is initialized with the value `$HOME/sample.fa`. This value can be overridden when launching the pipeline, by simply adding the option `--in <value>` to the script command line
- line 5: Defines a variable `sequences` holding a reference for the file whose name is specified by the `params.in` parameter
- line 6: Defines a variable `SPLIT` whose value is `gsplit` when the script is executed on a Mac OSX or `csplit` when it runs on Linux. This is the name of the tool that is used to split the file.
- lines 8-20: The process that splits the provided file.
- line 10: Opens the *input* declaration block. The lines following this clause are interpreted as input definitions.
- line 11: Defines the process input file. This file is received from the variable `sequences` and will be named `input.fa`.
- line 13: Opens the *output* declaration block. Lines following this clause are interpreted as output definitions.
- line 14: Defines that the process outputs files whose names match the pattern `seq_*`. These files are sent over the channel `records`.
- lines 16-18: The actual script executed by the process to split the provided file.
- lines 22-33: Defines the second process, that receives the splits produced by the previous process and reverses their content.

- line 24: Opens the *input* declaration block. Lines following this clause are interpreted as input definitions.
- line 25: Defines the process input file. This file is received through the channel `records`.
- line 27: Opens the *output* declaration block. Lines following this clause are interpreted as output definitions.
- line 28: The *standard output* of the executed script is declared as the process output. This output is sent over the channel `result`.
- lines 30-32: The actual script executed by the process to *reverse* the content of the received files.
- line 35: Prints a *result* each time a new item is received on the `result` channel.

Tip: The above example can manage only a single file at a time. If you want to execute it for two (or more) different files you will need to launch it several times.

It is possible to modify it in such a way that it can handle any number of input files, as shown below.

In order to make the above script able to handle any number of files simply replace *line 3* with the following line:

```
sequences = Channel.fromPath(params.in)
```

By doing this the `sequences` variable is assigned to the channel created by the *fromPath* method. This channel emits all the files that match the pattern specified by the parameter `params.in`.

Given that you saved the script to a file named `example.nf` and you have a list of *FASTA* files in a folder named `dataset/`, you can execute it by entering this command:

```
nextflow example.nf --in 'dataset/*.fa'
```

Warning: Make sure you enclose the `dataset/*.fa` parameter value in single-quotation characters, otherwise the BASH environment will expand the `*` symbol to the actual file names and the example won't work.

20.2 More examples

You can find at [this link](#) a collection of examples introducing Nextflow scripting.

Check also [Awesome Nextflow](#) for a list of pipelines developed by the Nextflow community.

21.1 How do I process multiple input files in parallel?

Q: I have a collection of input files (e.g. carrots.fa, onions.fa, broccoli.fa). How can I specify that a process is performed on each input file in a parallel manner?

A: The idea here is to create a channel that will trigger a process execution for each of your files. First define a parameter that specifies where the input files are:

```
params.input = "data/*.fa"
```

Each of the files in the data directory can be made into a channel with:

```
vegetable_datasets = Channel.fromPath(params.input)
```

From here, each time the variable `vegetable_datasets` is called as an input to a process, the process will be performed on each of the files in the vegetable datasets. For example, each input file may contain a collection of unaligned sequences. We can specify a process to align them as follows:

```
process clustalw2_align {
  input:
    file vegetable_fasta from vegetable_datasets

  output:
    file "${vegetable_fasta.baseName}.aln" into vegetable_alns

  script:
    """
    clustalw2 -INFILE=${vegetable_fasta}
    """
}
```

This would result in the alignment of the three vegetable fasta files into `carrots.aln`, `onions.aln` and `broccoli.aln`.

These aligned files are now in the channel `vegetable_alns` and can be used as input for a further process.

21.2 How do I get a unique ID based on the file name?

Q: How do I get a unique identifier based on a dataset file names (e.g. broccoli from broccoli.fa) and have the results going to a specific folder (e.g. results/broccoli/)?

A: First we can specify a results directory as shown below:

```
results_path = $PWD/results
```

The best way to manage this is to have the channel emit a tuple containing both the file base name (`broccoli`) and the full file path (`data/broccoli.fa`):

```
datasets = Channel
    .fromPath(params.input)
    .map { file -> tuple(file.baseName, file) }
```

And in the process we can then reference these variables (`datasetID` and `datasetFile`):

```
process clustalw2_align {
    publishDir "$results_path/$datasetID"

    input:
    set datasetID, file(datasetFile) from datasets

    output:
    set datasetID, file("${datasetID}.aln") into aligned_files

    script:
    """
    clustalw2 -INFILE=${datasetFile} -OUTFILE=${datasetID}.aln
    """
}
```

In our example above would now have the folder `broccoli` in the results directory which would contain the file `broccoli.aln`.

If the input file has multiple extensions (e.g. `broccoli.tar.gz`), you will want to use `file.simpleName` instead, to strip all of them (available since Nextflow 0.25+).

21.3 How do I use the same channel multiple times?

Q: Can a channel be used in two input statements? For example, I want carrots.fa to be aligned by both ClustalW and T-Coffee.

A: A channel can be consumed only by one process or operator (except if channel only ever contains one item). You must duplicate a channel before calling it as an input in different processes. First we create the channel emitting the input files:

```
vegetable_datasets = Channel.fromPath(params.input)
```

Next we can split it into two channels by using the *into* operator:

```
vegetable_datasets.into { datasets_clustalw; datasets_tcoffee }
```

Then we can define a process for aligning the datasets with *ClustalW*:

```
process clustalw2_align {
  input:
    file vegetable_fasta from datasets_clustalw

  output:
    file "${vegetable_fasta.baseName}.aln" into clustalw_alns

  script:
    """
    clustalw2 -INFILE=${vegetable_fasta}
    """
}
```

And a process for aligning the datasets with *T-Coffee*:

```
process tcoffee_align {
  input:
    file vegetable_fasta from datasets_tcoffee

  output:
    file "${vegetable_fasta.baseName}.aln" into tcoffee_alns

  script:
    """
    t_coffee ${vegetable_fasta}
    """
}
```

The upside of splitting the channels is that given our three unaligned fasta files (`broccoli.fa`, `onion.fa` and `carrots.fa`) six alignment processes (three x ClustalW) + (three x T-Coffee) will be executed as parallel processes.

21.4 How do I invoke custom scripts and tools?

Q: I have executables in my code, how should I call them in Nextflow?

A: Nextflow will automatically add the directory `bin` into the `PATH` environmental variable. So therefore any executable in the `bin` folder of a Nextflow pipeline can be called without the need to reference the full path.

For example, we may wish to reformat our *ClustalW* alignments from Question 3 into *PHYLIP* format. We will use the handy tool `esl-reformat` for this task.

First we place copy (or create a symlink to) the `esl-reformat` executable to the project's `bin` folder. From above we see the *ClustalW* alignments are in the channel `clustalw_alns`:

```
process phylip_reformat {
  input:
    file clustalw_alignment from clustalw_alns

  output:
    file "${clustalw_alignment.baseName}.phy" to clustalw_phylips

  script:
```

(continues on next page)

(continued from previous page)

```

    """
    esl-reformat phylip ${clustalw_alignment} ${clustalw_alignment.baseName}.phy
    """
}

process generate_bootstrap_replicates {
    input:
    file clustalw_phylip from clustalw_phylips
    output:
    file "${clustalw_alignment.baseName}.phy" to clustalw_phylips

    script:
    """
    esl-reformat phylip ${clustalw_alignment} ${clustalw_alignment.baseName}.phy
    """
}

```

21.5 How do I iterate over a process n times?

To perform a process n times, we can specify the input to be each x from $y..z$. For example:

```

bootstrapReplicates=100

process bootstrapReplicateTrees {
    publishDir "$results_path/$datasetID/bootstrapsReplicateTrees"

    input:
    each x from 1..bootstrapReplicates
    set val(datasetID), file(ClustalwPhylips)

    output:
    file "bootstrapTree_${x}.nwk" into bootstrapReplicateTrees

    script:
    // Generate Bootstrap Trees

    """
    raxmlHPC -m PROTGAMEJTT -n tmpPhylip${x} -s tmpPhylip${x}
    mv "RAxML_bestTree.tmpPhylip${x}" bootstrapTree_${x}.nwk
    """
}

```

21.6 How do I iterate over nth files from within a process?

Q: For example, I have 100 files emitted by a channel. I wish to perform one process where I iterate over each file inside the process.

A: The idea here is to transform a channel emitting multiple items into a channel that will collect all files into a list object and produce that list as a single emission. We do this using the `collect()` operator. The process script would then be able to iterate over the files by using a simple for-loop.

This is also useful if all the items of a channel are required to be in the work directory.

```
process concatenateBootstrapReplicates {
  publishDir "$results_path/$datasetID/concatenate"

  input:
  file bootstrapTreeList from bootstrapReplicateTrees.collect()

  output:
  file "concatenatedBootstrapTrees.nwk"

  // Concatenate Bootstrap Trees
  script:
  """
  for every treeFile in ${bootstrapTreeList}
  do
    cat \${treeFile} >> concatenatedBootstrapTrees.nwk
  done
  """
}
```

21.7 How do I use a specific version of Nextflow?

Q: I need to specify a version of Nextflow to use, or I need to pull a snapshot release.

A: Sometimes it is necessary to use a different version of Nextflow for a specific feature or testing purposes. Nextflow is able to automatically pull versions when the NXF_VER environment variable is defined on the commandline.

```
NXF_VER=0.28.0 nextflow run main.nf
```