

# GET STARTED WITH **nextflow**

Paolo Di Tommaso  
Bio in Docker Symposium - London

10 Nov 2015

these slides are published at this link

<http://www.nextflow.io/misc/bioindocker-tutorial.pdf>

# WHAT NEXTFLOW IS

- A computing runtime which executes Nextflow pipeline scripts
- A programming DSL that simplify writing of highly parallel computational pipelines reusing your existing scripts and tools

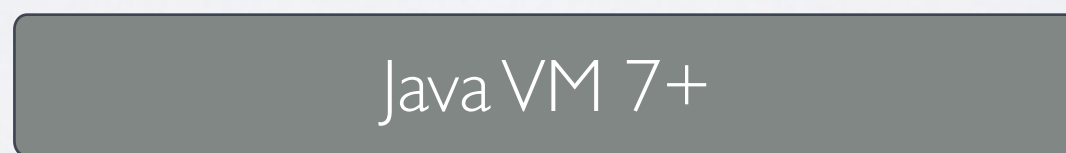
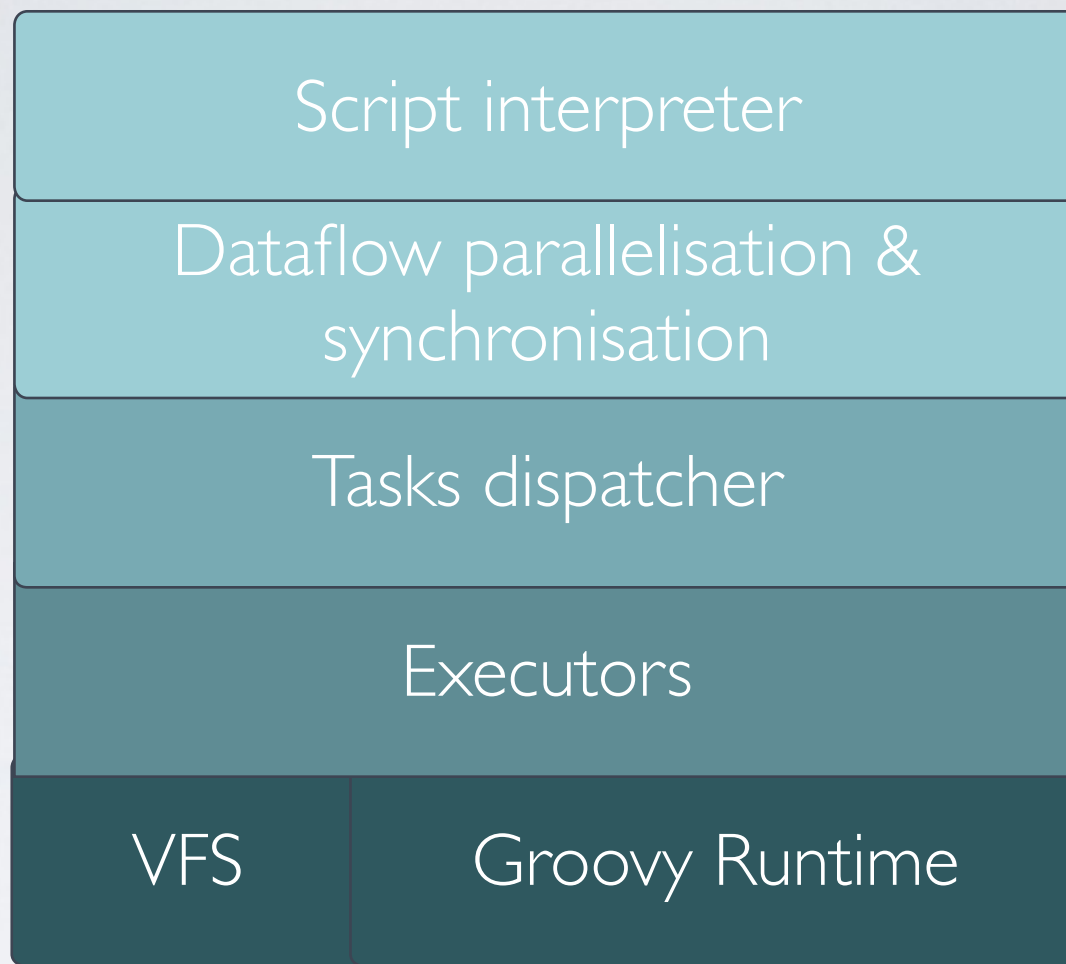
# NEXTFLOW DSL

- It is NOT a new programming language
- It extends the Groovy scripting language
- It provides a multi-paradigm programming environment



# MULTI-PARADIGM

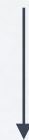
Imperative  
Object-oriented programming  
+  
Declarative concurrency  
Reactive-functional programming



# HOW TO INSTALL\*

Use the following command:

```
wget -qO- get.nextflow.io | bash
```



nextflow

\* It requires: Unix-like OS (Linux, OSX, etc) and Java 7/8

# GET STARTED

Install Nextflow:

```
$ wget -qO- get.nextflow.io | bash
```

- or -

```
$ curl -fsSL get.nextflow.io | bash
```



# DOWNLOAD EXAMPLES

Open a shell terminal and type:

```
$ git clone https://github.com/nextflow-io/examples.git  
$ cd examples  
$ echo 'docker.enabled=true' >> nextflow.config  
$ docker pull nextflow/examples
```

\* Mac OSX users:

make sure to work in a path under `/Users/<something>`

# THE BASIC

Variables and assignments

```
x = 1  
y = 10.5  
str = 'hello world!'  
p = x; q = y
```

# THE BASIC

Printing values

```
x = 1  
y = 10.5  
str = 'hello world!'
```

```
print x
```

```
print str
```

```
print str + '\n'
```

```
println str
```

# THE BASIC

Printing values

```
x = 1  
y = 10.5  
str = 'hello world!'
```

```
print(x)
```

```
print(str)
```

```
print(str + '\n')
```

```
println(str)
```

# MORE ON STRINGS

```
str = 'bioinformatics'  
print str[0]
```

```
print "$str is cool!"  
print "Current path: $PWD"
```

```
str = '''  
    multi  
    line  
    string  
    '''
```

```
str = """  
    User: $USER  
    Home: $HOME  
    """
```




# COMMON STRUCTURES & PROGRAMMING IDIOMS

- Data structures: Lists & Maps
- Control statements: if, for, while, etc.
- Functions and classes
- File I/O operations

# 6 PAGES PRIMER

#15  
Get More Refcardz! Visit [refcardz.com](http://refcardz.com)  
www.dzone.com  
Groovy



## Groovy

By Dierk König

**CONTENTS INCLUDE:**

- Groovy/Java Integration
- Language Elements
- Operators
- Collective Datatypes
- Meta-Programming
- Hot Tips and more...

### ABOUT GROOVY

Groovy is a dynamic language for the Java™ Virtual Machine (JVM). It shines with full object orientation, scriptability, optional typing, operator automation, lexical declarations for the most common data types, advanced concepts like closures and ranges, compact property syntax and seamless Java™ integration. This reference card provides exactly the kind of information you are likely to look up when programming Groovy.

### STARTING GROOVY

Install Groovy from <http://groovy.codehaus.org> and you will have the following commands available:

Command	Purpose
groovy	Execute Groovy code
groovyc	Compile Groovy code
groovysh	Open Groovy shell
groovyconsole	Open Groovy IDE console
groovydoc	Generate Groovy API documentation

The `groovy` command comes with `-h` and `-help` options to show all options and required arguments. Typical usages are:

Execute file `MyScript.groovy`

```
groovy MyScript
```

Evaluate `(e)` on the command line

```
groovy -e "print 12.5/Main.RV"
```

Print `(e)` for each line of input

```
echo 12.5 | groovy -e "line.toDouble() * Math.PI"
```

Inline `with()` file `data.txt` by reversing each line and save a backup

```
groovy -i bak -e "line.reverse()" data.txt
```

### GROOVY / JAVA INTEGRATION

From Groovy, you can call any Java code like you would do from Java. It's effortless.

From Java, you can call Groovy code in the following ways:

Note that you need to have the `groovy-all.jar` in your classpath.

**Cross-compilation**

Use `groovyc`, the `groovy-ant` or task or your IDE integration to compile your Groovy code together with your Java code. This enables you to use your Groovy code as if it was written in Java.

**Eval**

Use class `groovy.util.Eval` for evaluating simple code that is captured in a Java String. (WD) `Eval.eval("1.2.3.*4.5+4")`.

### GroovyShell

Use `groovy.util.GroovyShell` for more flexibility in the binding and optional pre-parsing.

```
GroovyShell shell = new GroovyShell();
Script script = shell.parse("y = x*x");
Binding binding = new Binding();
script.setBinding(binding);
binding.setVariable("x", 2);
script.run();
(int) binding.getVariable("y");
```

Chapter 11 of *Groovy in Action* has more details about integration options. Here is an overview:

Integration option	Features/properties
FullGroovyShell	For small expressions - including security
GroovyShellEngine	For dependent scripts - including classes, security
GroovyShellCode	For each of various - including security
SpringBeans	Integrate with Spring - including
WebGroovy	Any language script that has a <code>Servlet</code> API - including security, response, <code>Servlet</code>

### LANGUAGE ELEMENTS

#### Classes & Scripts

A Groovy class declaration looks like in Java. Default visibility modifier is `public`.

```
class MyClass {
    void myMethod(String argument) {}
}
```

### Get More Refcardz (They're free!)

- Authoritative content
- Designed for developers
- Written by top experts
- Latest tools & technologies
- Hot tips & examples
- Bonus content online
- New issue every 1-2 weeks

Subscribe Now for FREE!  
[Refcardz.com](http://Refcardz.com)

DZone, Inc. | [www.dzone.com](http://www.dzone.com)

<http://refcardz.dzone.com/refcardz/groovy>

# MAIN ABSTRACTIONS

- Processes: run any piece of script
- Channels: data streams that allows processes to communicate
- Operators: transform channels content

# CHANNELS

- It connects two processes/operators
- Asynchronous unidirectional FIFO queue
- Write operation is NOT blocking
- Read operation is blocking
- Once an item is read is removed from the queue



# CHANNELS

```
my_channel = Channel.create()
```

```
some_items = Channel.from(10, 20, 30, ..)
```

```
single_file = Channel.fromPath('some/file/name')
```

```
more_files = Channel.fromPath('some/data/path/*')
```



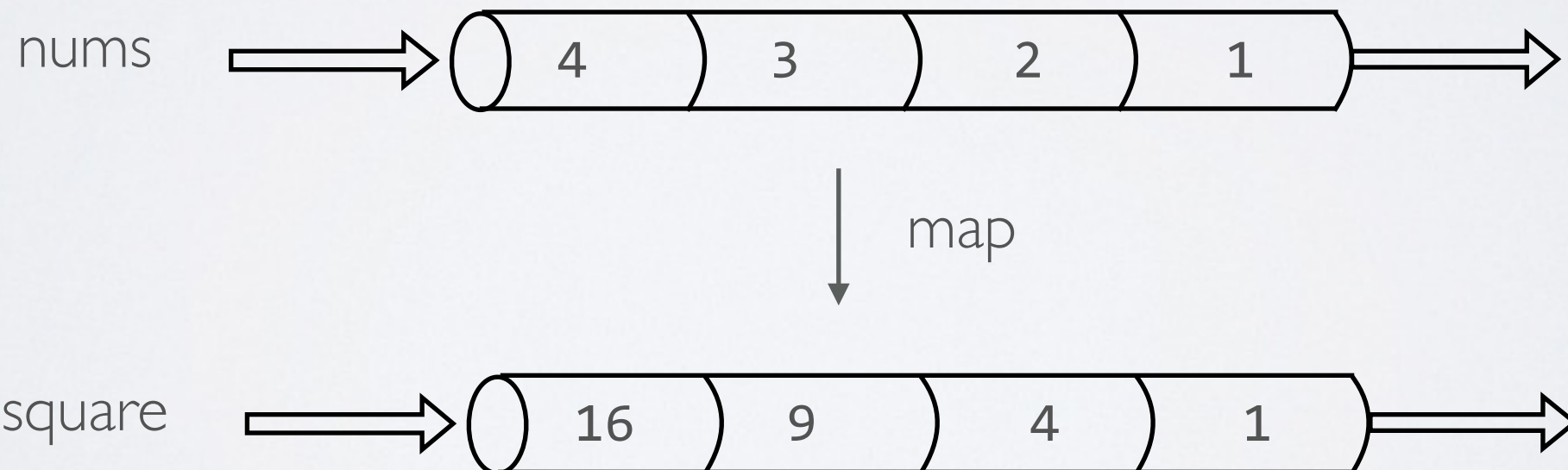


# OPERATORS

- Functions applied to channels
- Transform channels content
- Can be used also to filter, fork and combine channels
- Operators can be chained to implement custom behaviours

# OPERATORS

```
nums = Channel.from(1,2,3,4)  
square = nums.map { it -> it * it }
```



# OPERATORS CHAINING

```
Channel.from(1,2,3,4)
    .map { it -> [it, it*it] }
    .subscribe { num, sqr -> println "Square of: $num is $sqr" }
```

```
// it prints
Square of: 1 is 1
Square of: 2 is 4
Square of: 3 is 9
Square of: 4 is 16
```

# SPLIT FASTA FILE(S)

```
Channel.fromPath(' /some/path/fasta.fa')  
    .splitFasta()  
    .println()
```

```
Channel.fromPath(' /some/path/fasta.fa')  
    .splitFasta(by: 3)  
    .println()
```

```
Channel.fromPath(' /some/path/*.fa')  
    .splitFasta(by: 3)  
    .println()
```



# SPLITTING OPERATORS

You can split text object or files using the splitting methods:

- `splitText` - line by line
- `splitCsv` - comma separated values format
- `splitFasta` - by FASTA sequences
- `splitFastq` - by FASTQ sequences



# EXAMPLE I

- Split a FASTA file in sequence
- Parse a FASTA file and count number of sequences matching specified ID

# EXAMPLE I

```
$ nextflow run channel_split.nf
```

```
$ nextflow run channel_filter.nf
```

# PROCESS

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

```
process sayHello {
```

```
    input:  
    val str
```

```
    output:  
    stdout into result
```

```
    script:  
    """  
    echo $str world!  
    """
```

```
}
```

```
result.subscribe { print it }
```

# PROCESS INPUTS

```
process procName {  
    input:  
        <input type> <name> [from <source channel>] [attributes]  
  
    ""  
    <your script>  
    ""  
  
}
```

# PROCESS INPUTS

```
process procName {  
    input:  
        val  x from ch_1  
        file y from ch_2  
        file 'data.fa' from ch_3  
        stdin from from ch_4  
        set (x, 'file.txt') from ch_5  
  
    """"  
    <your script>  
    """"  
  
}
```



# PROCESS INPUTS

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

```
process blastThemAll {  
    input:  
    file 'query.fa' from proteins  
  
    """"  
    blastp -query query.fa -db nr  
    """"  
  
}
```

# PROCESS OUTPUTS

```
process randomNum {  
  
    output:  
    file 'result.txt' into numbers  
  
    '''  
    echo $RANDOM > result.txt  
    '''  
  
}  
  
numbers.println { "Received: " + it.text }
```

# EXAMPLE 2

- Execute a process running a BLAST job given an input file
- Execute a BLAST job printing the produced output

# EXAMPLE 2

```
$ nextflow run process_input.nf
```

```
$ nextflow run process_output.nf
```



# USE YOUR FAVOURITE LANG

```
process pyStuff {  
    """  
    #!/usr/bin/env python  
  
    x = 'Hello'  
    y = 'world!'  
    print "%s - %s" % (x,y)  
    """  
}
```

# PIPELINES PARAMETERS

Simply declares some variables prefixed by params

```
params.p1 = 'alpha'  
params.p2 = 'beta'  
:
```

When launching your script you can override  
the default values

```
$ nextflow run <script file> --p1 'delta' --p2 'gamma'
```

# EXAMPLE 3

Split a FASTA file and execute a BLAST query for each chunk

# EXAMPLE 3

```
$ nextflow run split_fasta.nf
```

```
$ nextflow run split_fasta.nf --chunkSize 2
```

```
$ nextflow run split_fasta.nf --chunkSize 2 --query data/p/*.fa
```



# COLLECT FILE

The operator `collectFile` allows to gather items produced by upstream processes

Collect all items to a single file

```
my_results.collectFile(name: 'result.txt')
```

# COLLECT FILE

The operator collectFile allows to gather items produced by upstream processes

Collect the items and group them into files having a names defined by a grouping criteria

```
my_items.collectFile(storeDir:'path/name') {  
  
    def key = get_a_key_from_the_item(it)  
    def content = get_the_item_value(it)  
    [ key, content ]  
  
}
```

# EXAMPLE 4

Execute many BLAST queries and gather the results to a single file

```
$ nextflow run split_and_collect.nf
```

# EXAMPLE 5

Gather results using a process a user process

```
$ nextflow run split_and_gather.nf
```



# EXAMPLE 6

Groups read-pairs and process them using a process

```
$ nextflow run group_read_pairs.nf
```

# CONFIG FILE

- Pipeline configuration can be externalised to a file named `nextflow.config`
  - parameters
  - environment variables
  - required resources (mem, cpus, queue, etc)
  - modules/containers
  - execution profiles

# CONFIG FILE

```
params.p1 = 'alpha'
```

```
params.p2 = 'beta'
```

```
env.VAR_1 = 'some_value'
```

```
env.CACHE_4_TCOFFEE = '/some/path/cache'
```

```
env.LOCKDIR_4_TCOFFEE = '/some/path/lock'
```

```
process.executor = 'sge'
```

# CONFIG FILE

Alternate syntax (almost) equivalent

```
params {  
    p1 = 'alpha'  
    p2 = 'beta'  
}
```

```
env {  
    VAR_1 = 'some_value'  
    CACHE_4_TCOFFEE = '/some/path/cache'  
    LOCKDIR_4_TCOFFEE = '/some/path/lock'  
}
```

```
process {  
    executor = 'sge'  
}
```

# HOW USE DOCKER

Specify in the config file the Docker image to use

```
process {  
    container = 'your/image:tag'  
}
```

Add the with-docker option when launching it

```
$ nextflow run <script name> -with-docker
```



# MULTIPLE CONTAINERS

Specify in the config file the Docker image to use

```
process {  
  
    $foo {  
        container = 'your/image:tag'  
    }  
  
    $bar {  
        container = 'another/image:latest'  
    }  
  
}
```

# HOW USE A CLUSTER

Define the CRG executor in `nextflow.config`

```
// default properties for any process
process {
    executor = 'sge' // other: lsf, pbs, slurm
    queue = 'short'
    cpus = 2
    memory = '4GB'
    scratch = true
}
```

# PROCESS RESOURCES

```
// default properties for any process
```

```
process {  
    executor = 'sge'  
    queue = 'short'  
    scratch = true  
}
```

```
// cpus for process 'foo'
```

```
process.$foo.cpus = 2
```

```
// resources for 'bar'
```

```
process.$bar.queue = 'long'
```

```
process.$bar.cpus = 4
```

```
process.$bar.memory = '4GB'
```

# CONFIG PROFILES

```
profiles {  
  
    standard {  
        process.executor = 'local'  
    }  
  
    crg {  
        process.executor = 'sge'  
        process.queue = 'long'  
        process.memory = '10GB'  
        process.container = 'image/name'  
    }  
  
    ebi {  
        process.executor = 'lsf'  
        process.queue = 'bio16'  
        process.module = 'ncbi-blast/2.2.27'  
    }  
  
}
```

# CONFIG PROFILES

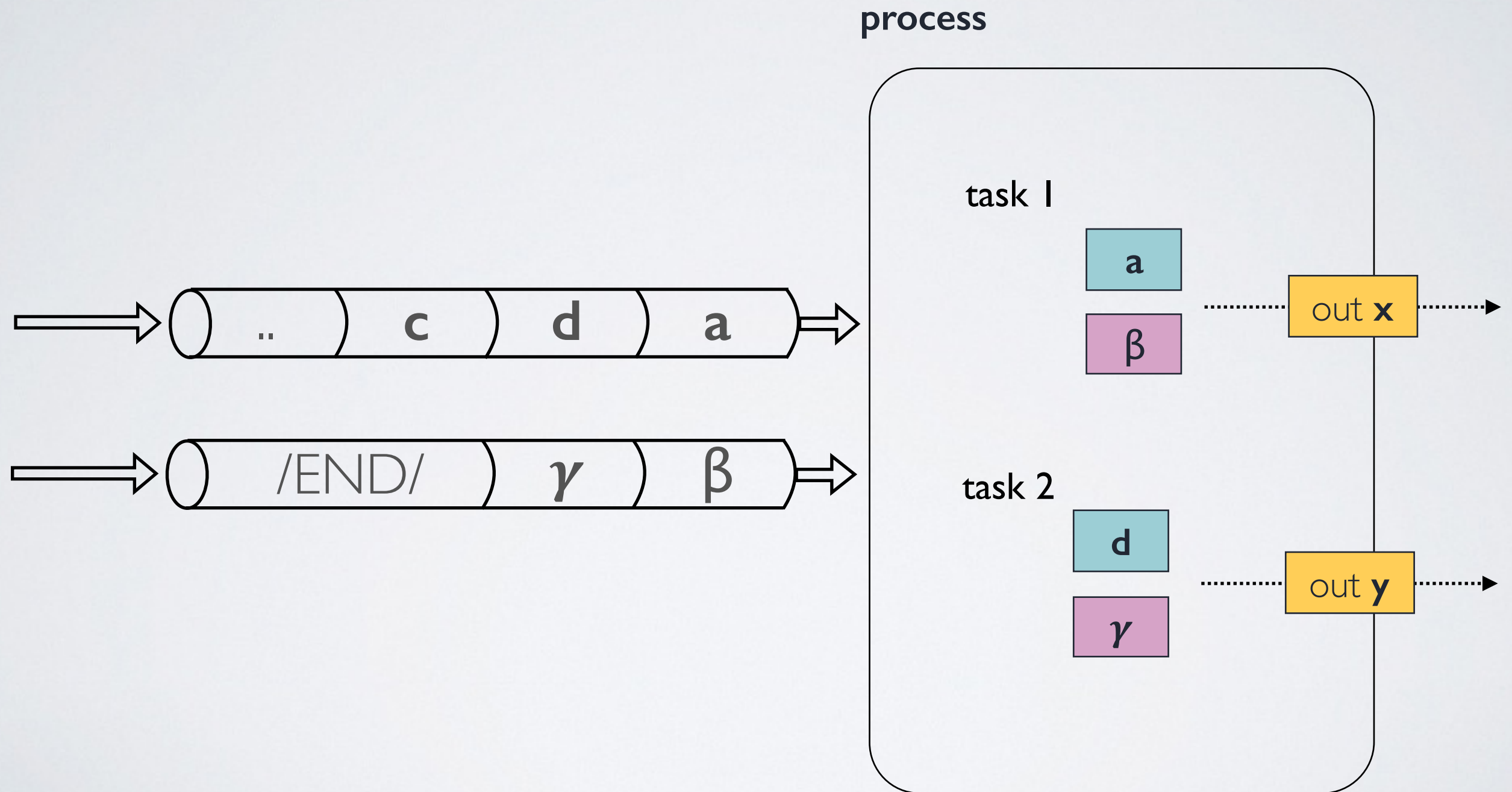
```
$ nextflow run <project> -profile crg|ebi
```



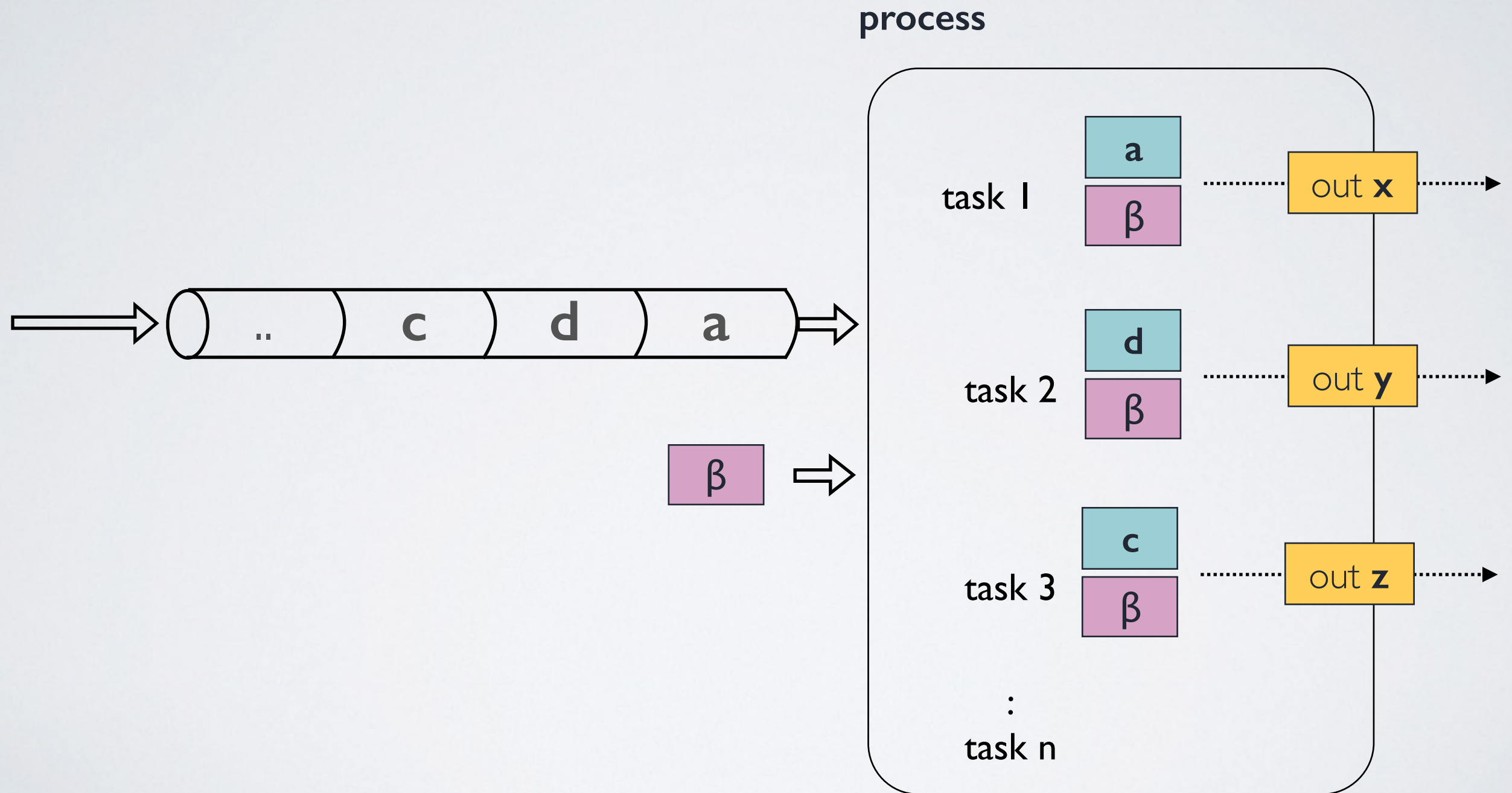
# PUBLISH YOUR PIPELINE

- Create a Github repo (or Gitlab/BitBucket)
- Name the pipeline script `main.nf`
- Create a `nextflow.config` file
- Put other scrips into a folder named `bin`
- Create a Docker image for binary dependencies

# UNDERSTANDING MULTIPLE INPUTS



# UNDERSTANDING MULTIPLE INPUTS



# LINKS

project home

<http://nextflow.io>

tutorials

<https://github.com/nextflow-io/examples>

chat

<https://gitter.im/nextflow-io/nextflow>