Dashboard

Get insights into your workflows with the Dashboard.

The first time you access the main **Dashboard**, you'll see the **Welcome Page** and you can click **Create my first flow** to launch a Guided Tour.

Once you have executed a flow, you will see your flow executions in the dashboard.

The Dashboard provides a load of useful data right at your finger tips, including: - Executions over time - Execution Status for Today, Yesterday as well as Last 30 days - Executions per namespace - Execution errors per namespace - List of failed Executions - List of error logs

The dashboard is interactive so you can click on any element to get more information about it.

Figure 1: UI

Flowable Tasks

Control your orchestration logic.

Flowable tasks control the orchestration logic — run tasks or subflows in parallel, create loops and conditional branching.

Flowable Tasks don't run heavy operations — those are handled by workers.

Flowable Tasks are used for branching, grouping, running tasks in parallel, and more.

Flowable Tasks use expressions from the execution context to define the next tasks to run. For example, you can use the outputs of a previous task in a `Switch` task to decide which task to run next.

Sequential

This task processes tasks one after another sequentially. It is used to group tasks.

```yaml
id: sequential
namespace: company.team

tasks:
  - id: sequential
    type: io.kestra.plugin.core.flow.Sequential
    tasks:
      - id: 1st
        type: io.kestra.plugin.core.debug.Return
        format: "{{ task.id }} > {{ taskrun.startDate }}"

      - id: 2nd
        type: io.kestra.plugin.core.debug.Return
        format: "{{ task.id }} > {{ taskrun.id }}"

  - id: last
    type: io.kestra.plugin.core.debug.Return
    format: "{{ task.id }} > {{ taskrun.startDate }}"
```

::alert{type="info"} You can access the output of a sibling task using the syntax `{{ outputs.sibling.value }}`. ::

::next-link Sequential Task documentation ::

Parallel

This task processes tasks in parallel. It makes it convenient to process many tasks at once.

```yaml
id: parallel
namespace: company.team

tasks:
  - id: parallel
    type: io.kestra.plugin.core.flow.Parallel
    tasks:
      - id: 1st
        type: io.kestra.plugin.core.debug.Return
        format: "{{ task.id }} > {{ taskrun.startDate }}"

      - id: 2nd
        type: io.kestra.plugin.core.debug.Return
        format: "{{ task.id }} > {{ taskrun.id }}"

  - id: last
    type: io.kestra.plugin.core.debug.Return
    format: "{{ task.id }} > {{ taskrun.startDate }}"
```

::alert{type="warning"} You cannot access the output of a sibling task as tasks will be run in parallel. ::

::next-link Parallel Task documentation ::

Switch

This task processes a set of tasks conditionally depending on a contextual variable's value.

In the following example, an input will be used to decide which task to run next.

```yaml
id: switch
namespace: company.team

inputs:
  - id: param
    type: BOOLEAN

tasks:
```

```
  - id: decision
    type: io.kestra.plugin.core.flow.Switch
    value: "{{ inputs.param }}"
    cases:
      true:
        - id: is_true
          type: io.kestra.plugin.core.log.Log
          message: "This is true"
      false:
        - id: is_false
          type: io.kestra.plugin.core.log.Log
          message: "This is false"
```

::next-link Switch Task documentation ::

If

This task processes a set of tasks conditionally depending on a condition. The condition must coerce to a boolean. Boolean coercion allows 0, -0, null and '' to coerce to false, all other values to coerce to true. The `else` branch is optional.

In the following example, an input will be used to decide which task to run next.

```
id: if_condition
namespace: company.team

inputs:
  - id: param
    type: BOOLEAN

tasks:
  - id: if
    type: io.kestra.plugin.core.flow.If
    condition: "{{ inputs.param }}"
    then:
      - id: when_true
        type: io.kestra.plugin.core.log.Log
        message: "This is true"
    else:
      - id: when_false
        type: io.kestra.plugin.core.log.Log
        message: "This is false"
```

::next-link If Task documentation ::

ForEach

This task will execute a group of tasks for each value in the list.

In the following example, the variable is static, but it can be generated from a previous task output and starts an arbitrary number of subtasks.

```yaml
id: foreach_example
namespace: company.team

tasks:
  - id: for_each
    type: io.kestra.plugin.core.flow.ForEach
    values: ["value 1", "value 2", "value 3"]
    tasks:
      - id: before_if
        type: io.kestra.plugin.core.debug.Return
        format: "Before if {{ taskrun.value }}"
      - id: if
        type: io.kestra.plugin.core.flow.If
        condition: '{{ taskrun.value == "value 2" }}'
        then:
          - id: after_if
            type: io.kestra.plugin.core.debug.Return
            format: "After if {{ parent.taskrun.value }}"
```

::alert{type="info"} You can access the output of a sibling task using the syntax `{{ outputs.sibling[taskrun.value].value }}`. ::

This example shows how to run tasks in parallel for each value in the list. All child tasks of the parallel task will run in parallel. However, due to the `concurrencyLimit` property set to 2, only two parallel task groups will run at any given time.

```yaml
id: parallel_tasks_example
namespace: company.team

tasks:
  - id: for_each
    type: io.kestra.plugin.core.flow.ForEach
    values: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
    concurrencyLimit: 2
    tasks:
      - id: parallel
        type: io.kestra.plugin.core.flow.Parallel
        tasks:
        - id: log
          type: io.kestra.plugin.core.log.Log
          message: Processing {{ parent.taskrun.value }}
        - id: shell
          type: io.kestra.plugin.scripts.shell.Commands
          commands:
```

```
      - sleep {{ parent.taskrun.value }}
```

For processing items, or forwarding processing to a subflow, ForEachItem is better suited.

::next-link ForEach Task documentation ::

ForEachItem

This task allows you to iterate over a list of items and run a subflow for each item, or for each batch containing multiple items.

Syntax:

```yaml
  - id: each
    type: io.kestra.plugin.core.flow.ForEachItem
    items: "{{ inputs.file }}" # could be also an output variable {{ outputs.extract.uri }}
    inputs:
      file: "{{ taskrun.items }}" # items of the batch
    batch:
      rows: 4
    namespace: company.team
    flowId: subflow
    revision: 1 # optional (default: latest)
    wait: true # wait for the subflow execution
    transmitFailed: true # fail the task run if the subflow execution fails
    labels: # optional labels to pass to the subflow to be executed
      key: value
```

This will execute the subflow `company.team.subflow` for each batch of items. To pass the batch of items to a subflow, you can use inputs. The example above uses an input of `FILE` type called `file` that takes the URI of an internal storage file containing the batch of items.

The next example shows you how you can access the outputs from each subflow executed. The ForEachItem automatically merges the URIs of the outputs from each subflow into a single file. The URI of this file is available through the `subflowOutputs` output.

```yaml
id: for_each_item
namespace: company.team

tasks:
  - id: generate
    type: io.kestra.plugin.scripts.shell.Script
    script: |
      for i in $(seq 1 10); do echo "$i" >> data; done
    outputFiles:
      - data
```

```yaml
  - id: for_each_item
    type: io.kestra.plugin.core.flow.ForEachItem
    items: "{{ outputs.generate.outputFiles.data }}"
    batch:
      rows: 4
    wait: true
    flowId: my_subflow
    namespace: company.team
    inputs:
       value: "{{ taskrun.items }}"

  - id: for_each_outputs
    type: io.kestra.plugin.core.log.Log
    message: "{{ outputs.forEachItem_merge.subflowOutputs }}" # Log the URI of the file con
```

::next-link ForEachItem Task documentation ::

`ForEach` vs `ForEachItem` Both `ForEach` and `ForEachItem` are similar, but there are specific use cases that suit one over the other: - `ForEach` generates a lot of Task Runs which can impact performance. - `ForEachItem` generates separate executions using Subflows for the group of tasks. This scales better for larger datasets.

Read more about performance optimization here.

AllowFailure

This task will allow child tasks to fail. If any child task fails: - The AllowFailure failed task will be marked as status `WARNING`. - All children's tasks inside the AllowFailure will be stopped immediately. - The Execution will continue for all others tasks. - At the end, the execution as a whole will also be marked as status `WARNING`.

In the following example: - `allow_failure` will be labelled as `WARNING`. - `ko` will be labelled as `FAILED`. - `next` will not be run. - `end` will be run and labelled `SUCCESS`.

```yaml
id: each
namespace: company.team

tasks:
  - id: allow_failure
    type: io.kestra.plugin.core.flow.AllowFailure
    tasks:
      - id: ko
```

```
        type: io.kestra.plugin.core.execution.Fail
      - id: next
        type: io.kestra.plugin.core.debug.Return
        format: "{{ task.id }} > {{ taskrun.startDate }}"

  - id: end
    type: io.kestra.plugin.core.debug.Return
    format: "{{ task.id }} > {{ taskrun.startDate }}"
```

::next-link AllowFailure Task documentation ::

Fail

This task will fail the flow; it can be used with or without conditions.

Without conditions, it can be used, for example, to fail on some switch value.

```
id: fail_on_switch
namespace: company.team

inputs:
  - id: param
    type: STRING
    required: true

tasks:
  - id: switch
    type: io.kestra.plugin.core.flow.Switch
    value: "{{ inputs.param }}"
    cases:
      case1:
        - id: case1
          type: io.kestra.plugin.core.log.Log
          message: Case 1
      case2:
        - id: case2
          type: io.kestra.plugin.core.log.Log
          message: Case 2
      notexist:
        - id: fail
          type: io.kestra.plugin.core.execution.Fail
      default:
        - id: default
          type: io.kestra.plugin.core.log.Log
          message: default
```

With conditions, it can be used, for example, to validate inputs.

```yaml
id: fail_on_condition
namespace: company.team

inputs:
  - id: param
    type: STRING
    required: true

tasks:
  - id: before
    type: io.kestra.plugin.core.log.Log
    message: "I'm before the fail on condition"
  - id: fail
    type: io.kestra.plugin.core.execution.Fail
    condition: "{{ inputs.param == 'fail' }}"
  - id: after
    type: io.kestra.plugin.core.log.Log
    message: "I'm after the fail on condition"
```

::next-link Fail Task documentation ::

Subflow

This task will trigger another flow. This allows you to decouple the first flow from the second and monitor each flow individually.

You can pass flow outputs as inputs to the triggered subflow (those must be declared in the subflow).

```yaml
id: subflow
namespace: company.team

tasks:
  - id: "subflow"
    type: io.kestra.plugin.core.flow.Subflow
    namespace: company.team
    flowId: my-subflow
    inputs:
      file: "{{ inputs.myFile }}"
      store: 12
```

::next-link Subflow Task documentation ::

Worker

The `Worker` task is deprecated in favor of the`WorkingDirectory` task. The next section explains how you can use the`WorkingDirectory` task in order to allow multiple tasks to share a file system during the flow's Execution.

WorkingDirectory

By default, Kestra will launch each task in a new working directory, possibly on different workers if multiple ones exist.

The example below will run all tasks nested under the `WorkingDirectory` task sequentially. All those tasks will be executed in the same working directory, allowing the reuse of the previous tasks' output files in the downstream tasks. In order to share a working directory, all tasks nested under the `WorkingDirectory` task will be launched on the same worker.

This task can be particularly useful for compute-intensive file system operations.

```yaml
id: working_dir_flow
namespace: company.team

tasks:
  - id: working_dir
    type: io.kestra.plugin.core.flow.WorkingDirectory
    tasks:
      - id: first
        type: io.kestra.plugin.scripts.shell.Commands
        taskRunner:
          type: io.kestra.plugin.core.runner.Process
        commands:
          - 'echo "{{ taskrun.id }}" > {{ workingDir }}/stay.txt'

      - id: second
        type: io.kestra.plugin.scripts.shell.Commands
        taskRunner:
          type: io.kestra.plugin.core.runner.Process
        commands:
          - |
            echo '::{"outputs": {"stay":"'$(cat {{ workingDir }}/stay.txt)'"}}::'
```

This task can also cache files inside the working directory, for example, to cache script dependencies like the `node_modules` of a node `Script` task.

```yaml
id: node_with_cache
namespace: company.team

tasks:
  - id: working_dir
    type: io.kestra.plugin.core.flow.WorkingDirectory
    cache:
      patterns:
        - node_modules/**
      ttl: PT1H
```

```
    tasks:
      - id: script
        type: io.kestra.plugin.scripts.node.Script
        beforeCommands:
          - npm install colors
        script: |
          const colors = require("colors");
          console.log(colors.red("Hello"));
```

This task can also fetch files from namespace files and make them available to all child tasks.

```
id: node_with_cache
namespace: company.team

tasks:
  - id: working_dir
    type: io.kestra.plugin.core.flow.WorkingDirectory
    namespaceFiles:
      enabled: true
      include:
        - dir1/*.*
      exclude:
        - dir2/*.*
    tasks:
      - id: shell
        type: io.kestra.plugin.scripts.shell.Commands
        commands:
          - cat dir1/file1.txt
```

::next-link WorkingDirectory Task documentation ::

Pause

Kestra flows run until all tasks complete, but sometimes you need to: - Add a manual validation before continuing the execution. - Wait for some duration before continuing the execution.

For this, you can use the Pause task.

On the following example, the `validation` will pause until a manual modification of the task step, and the `wait` will wait for 5 minutes.

```
id: pause
namespace: company.team

tasks:
  - id: validation
    type: io.kestra.plugin.core.flow.Pause
```

```
    tasks:
      - id: ok
        type: io.kestra.plugin.scripts.shell.Commands
        taskRunner:
          type: io.kestra.plugin.core.runner.Process
        commands:
          - 'echo "started after manual validation"'

  - id: wait
    type: io.kestra.plugin.core.flow.Pause
    delay: PT5M
    tasks:
      - id: waited
        type: io.kestra.plugin.scripts.shell.Commands
        taskRunner:
          type: io.kestra.plugin.core.runner.Process
        commands:
          - 'echo "start after 5 minutes"'
```

::alert{type="info"} A Pause task without delay will wait indefinitely until the task state is changed to **Running**. For this: go to the **Gantt** tab of the **Execution** page, click on the task, select **Change status** on the contextual menu and select **Mark as RUNNING** on the form. This will make the task run until its end. ::

::next-link Pause Task documentation ::

DAG

This task allows defining dependencies between tasks by creating a directed acyclic graph (DAG). Instead of an explicit DAG structure, this task allows you to only define upstream dependencies for each task using the `dependsOn` property. This way, you can set dependencies more implicitly for each task, and Kestra will figure out the overall flow structure.

```
id: dag
namespace: company.team
tasks:
  - id: dag
    description: "my task"
    type: io.kestra.plugin.core.flow.Dag
    tasks:
      - task:
          id: task1
          type: io.kestra.plugin.core.log.Log
          message: I'm the task 1
      - task:
```

```
      id: task2
      type: io.kestra.plugin.core.log.Log
      message: I'm the task 2
    dependsOn:
      - task1
  - task:
      id: task3
      type: io.kestra.plugin.core.log.Log
      message: I'm the task 3
    dependsOn:
      - task1
  - task:
      id: task4
      type: io.kestra.plugin.core.log.Log
      message: I'm the task 4
    dependsOn:
      - task2
  - task:
      id: task5
      type: io.kestra.plugin.core.log.Log
      message: I'm the task 5
    dependsOn:
      - task4
      - task3
```

::next-link Dag Task documentation ::

Template (deprecated)

Templates are lists of tasks that can be shared between flows. You can define a template and call it from other flows, allowing them to share a list of tasks and keep these tasks updated without changing your flow.

The following example uses the Template task to use a template.

```
id: template
namespace: company.team

tasks:
  - id: template
    type: io.kestra.plugin.core.flow.Template
    namespace: company.team
    templateId: template
```

Programming Languages

Kestra is language agnostic. Use any programming language inside of your workflows.

Kestra works with any programming language with some having dedicated plugins to make it easier to use as well as libraries to make sending outputs and metrics back to Kestra easy.

Dedicated Plugins

Kestra currently supports the following programming languages with their own dedicated plugins:

1. Python
2. R
3. Node.js
4. Shell
5. Powershell
6. Julia
7. Ruby

Each of them have the following subgroup of plugins: - **Commands**: Execute scripts from a command line interface (good for longer files that may be written separately) - **Script**: Write your code directly inside of your YAML (good for short scripts that don't need a dedicated file)

Script Example

In this example, the Python script is inline:

```yaml
id: myflow
namespace: company.team

tasks:
  - id: "script"
    type: "io.kestra.plugin.scripts.python.Script"
    script: |
      from kestra import Kestra
      import requests
```

```
    response = requests.get('https://google.com')
    print(response.status_code)

    Kestra.outputs({'status': response.status_code, 'text': response.text})
  beforeCommands:
    - pip install requests kestra
```

Commands Example

In this example, the shell task is running dedicated commands, similar to how you would inside of your terminal.

```
id: myflow
namespace: company.team

tasks:
  - id: "commands"
    type: "io.kestra.plugin.scripts.shell.Commands"
    outputFiles:
      - first.txt
      - second.txt
    commands:
      - echo "1" >> first.txt
      - echo "2" >> second.txt
```

Run any language using the `Shell` task

Using `Commands`, you can run arbitrary commands in a Docker container. This means that you can use other languages as long as: 1. Their dependencies can be packaged into a Docker image 2. Their execution can be triggered from a `Shell` command line.

Below are a number of examples showing how you can do this with different programming languages.

For handling outputs and metrics, you can use the same approach that the `Shell` task uses by using `::{}::` syntax in log messages. Read more about it here.

Go

Here is an example flow that runs a Go file inside of a container using a `golang` image:

```
id: golang
namespace: company.team
```

golang_output

Figure 1: golang_output

```yaml
tasks:
  - id: go
    type: io.kestra.plugin.scripts.shell.Commands
    taskRunner:
      type: io.kestra.plugin.scripts.runner.docker.Docker
    containerImage: golang:latest
    namespaceFiles:
      enabled: true
    commands:
      - go run hello_world.go
```

The Go code is saved as a namespace file called `hello_world.go`:

```go
package main

import "fmt"

func main() {
    fmt.Println("Hello, World!")
}
```

When executed, we can see the print statement in the Kestra logs:

Check out the full guide which includes using outputs and metrics.

Rust

Here is an example flow that runs a Rust file inside of a container using a `rust` image:

```yaml
id: rust
namespace: company.team

tasks:
  - id: rust
    type: io.kestra.plugin.scripts.shell.Commands
    taskRunner:
      type: io.kestra.plugin.scripts.runner.docker.Docker
    containerImage: rust:latest
    namespaceFiles:
      enabled: true
    commands:
      - rustc hello_world.rs
      - ./hello_world
```

rust_output

Figure 2: rust_output

The Rust code is saved as a namespace file called `hello_world.rs`:

```
fn main() {
    println!("Hello, World!");
}
```

When executed, we can see the print statement in the Kestra logs:

Check out the full guide which includes using outputs and metrics.

Java

You can build custom plugins in Java which will allow you to add custom tasks to your workflows. If you're looking to execute something simpler, you can use the `Shell` task with a Docker container.

Here is an example flow that runs a Java file inside of a container using a `eclipse-temurin` image:

```
id: java
namespace: company.team

tasks:
  - id: java
    type: io.kestra.plugin.scripts.shell.Commands
    taskRunner:
      type: io.kestra.plugin.scripts.runner.docker.Docker
    containerImage: eclipse-temurin:latest
    namespaceFiles:
      enabled: true
    commands:
      - javac HelloWorld.java
      - java HelloWorld
```

The Java code is saved as a namespace file called `HelloWorld.java`:

```
class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

When executed, we can see the print statement in the Kestra logs:

java_output

Figure 3: java_output

c_output

Figure 4: c_output

C

Here is an example flow that runs a C file inside of a container using a `gcc` image:

```yaml
id: c
namespace: company.team

tasks:
  - id: c
    type: io.kestra.plugin.scripts.shell.Commands
    taskRunner:
      type: io.kestra.plugin.scripts.runner.docker.Docker
    containerImage: gcc:latest
    namespaceFiles:
      enabled: true
    commands:
      - gcc hello_world.c
      - ./a.out
```

The C code is saved as a namespace file called `hello_world.c`:

```c
#include <stdio.h>

int main() {
   printf("Hello, World!");
   return 0;
}
```

When executed, we can see the print statement in the Kestra logs:

C++

Here is an example flow that runs a C++ file inside of a container using a `gcc` image:

```yaml
id: cplusplus
namespace: company.team

tasks:
  - id: cpp
```

cpp_output

Figure 5: cpp_output

```
    type: io.kestra.plugin.scripts.shell.Commands
    taskRunner:
      type: io.kestra.plugin.scripts.runner.docker.Docker
    containerImage: gcc:latest
    namespaceFiles:
      enabled: true
    commands:
      - g++ hello_world.cpp
      - ./a.out
```

The C++ code is saved as a namespace file called `hello_world.cpp`:

```
#include <iostream>

int main() {
    std::cout << "Hello World!";
    return 0;
}
```

When executed, we can see the print statement in the Kestra logs:

TypeScript

You can execute TypeScript using the NodeJS plugin. To do so, we'll need to install TypeScript and compile our code to JavaScript using `tsc`.

Once we've done this, we can then execute it as we normally would with NodeJS. However, do note that our file is now a `.js` file.

```
id: typescript
namespace: company.team

tasks:
  - id: ts
    type: io.kestra.plugin.scripts.node.Commands
    namespaceFiles:
      enabled: true
    commands:
      - npm i -D typescript
      - npx tsc example.ts
      - node example.js
```

This example can be found in the Node.js docs. We're going to save it as `example.ts`.

ts_output

Figure 6: ts_output

```
type User = {
  name: string;
  age: number;
};

function isAdult(user: User): boolean {
  return user.age >= 18;
}

const justine: User = {
  name: 'Justine',
  age: 23,
};

const isJustineAnAdult: boolean = isAdult(justine);
console.log(isJustineAnAdult)
```

When executed, we can see the print statement in the Kestra logs:

You can read more about Node.js with TypeScript here.

PHP

Here is an example flow that runs a PHP file inside of a container using a `php` image:

```
id: php
namespace: company.team

tasks:
  - id: php
    type: io.kestra.plugin.scripts.shell.Commands
    taskRunner:
      type: io.kestra.plugin.scripts.runner.docker.Docker
    containerImage: php:8.4-rc-alpine
    namespaceFiles:
      enabled: true
    commands:
      - php hello_world.php
```

The PHP code is saved as a namespace file called `hello_world.php`:

```
<?php
echo "Hello, World!";
```

php_output

Figure 7: php_output

scala_output

Figure 8: scala_output

```
?>
```

When executed, we can see the print statement in the Kestra logs:

Scala

Here is an example flow that runs a Scala file inside of a container using a `sbtscala/scala-sbt` image:

```
id: scala
namespace: company.team

tasks:
  - id: scala
    type: io.kestra.plugin.scripts.shell.Commands
    taskRunner:
      type: io.kestra.plugin.scripts.runner.docker.Docker
    containerImage: sbtscala/scala-sbt:eclipse-temurin-17.0.4_1.7.1_3.2.0
    namespaceFiles:
      enabled: true
    commands:
      - scalac HelloWorld.scala
      - scala HelloWorld
```

The Scala code is saved as a namespace file called `HelloWorld.scala`:

```
object HelloWorld {
    def main(args: Array[String]) = {
        println("Hello, World!")
    }
}
```

When executed, we can see the print statement in the Kestra logs:

Perl

Here is an example flow that runs a Perl file inside of a container using a `perl` image:

```
id: perl
namespace: company.team
```

perl_output

Figure 9: perl_output

```yaml
tasks:
  - id: perl
    type: io.kestra.plugin.scripts.shell.Commands
    taskRunner:
      type: io.kestra.plugin.scripts.runner.docker.Docker
    containerImage: perl:5.41.2
    namespaceFiles:
      enabled: true
    commands:
      - perl hello_world.pl
```

The Perl code is saved as a namespace file called `hello_world.pl`:

```
#!/usr/bin/perl
use warnings;
print("Hello, World!\n");
```

When executed, we can see the print statement in the Kestra logs:

Software and Hardware Requirements

This page describes the software and hardware requirements for Kestra.

Software requirements

The table below lists the software requirements for Kestra.

Java

Kestra Edition	Required version	Note
OSS/Enterprise	>= 21 && < 22	Default 21 using Eclipse Temurin

Queue and Repository

Kestra Open-Source edition supports either Postgres or MySQL for use with the queue and repository components.

With Kestra Enterprise Edition (EE) you have the choice: - The same JDBC configuration as Open-Source for smaller scale deployments with an additional option to use SQL Server JDBC backend - Kafka plus Elasticsearch/Opensearch for large scale deployments

Kestra Edition	Database	Required version	Note
OSS/Enterprise	**PostgreSQL**	>=14 && <= 16.3	Default `latest`
OSS/Enterprise	**MySQL**	>=8 with exception 8.0.31	Default 8.3.2
Enterprise	**SQL Server (Preview)**	>= SQL Server 2019	Default 2022
Enterprise	**Apache Kafka**	>=3	
Enterprise	**Elasticsearch**	>=7	
Enterprise	**Opensearch**	>=2	

Internal Storage

Kestra Edition	Storage Provider	Required version	Note
OSS/Enterprise	MinIO	>=8	
OSS/Enterprise	Google Cloud GCS	N/A	
OSS/Enterprise	AWS S3	N/A	
OSS/Enterprise	Azure Blob Storage	N/A	

Hardware requirements

Kestra standalone server needs at least 4GiB of memory and 2vCPU to run correctly. In order to use script tasks, the server also needs to be able to run Docker-in-Docker (*this is why e.g. AWS ECR Fargate is currently not supported*).

If you need more guidance on how much memory and CPU to allocate to each architecture component, reach out to us and we'll help you with the sizing based on your expected workload.

Setup for Plugin Development

Setup your environment for Plugin Development.

Plugin Template

To get started with building a new plugin, make sure to use the plugin-template, as it comes prepackaged with the standardized repository structure and deployment workflows.

That template will create a project hosting a group of plugins — we usually create multiple subplugins for a given service. For example, there's only one plugin for AWS, but it includes many subplugins for specific AWS services.

::alert{type="warning"} Note that the Kestra plugin library **version** must align with your Kestra instance. You may encounter validation issues during flow creation (e.g. `Invalid bean` response with status 422) when some plugins are on an older version of the Kestra plugin library. In that case, you may want to update the file `plugin-yourplugin/gradle.properties` and set the `version` property to the correct Kestra version e.g.:

```
version=0.17.0-SNAPSHOT
kestraVersion=[0.17,)
```

It's not mandatory that your plugin version matches the Kestra version, Kestra's official plugins version will always match the minor version of Kestra but it's only a best practice.

Then rebuild and publish the plugin. ::

Requirements

Kestra plugins development requirements are: * Java 21 or later. * IntelliJ IDEA (or any other Java IDE, we provide only help for IntelliJ IDEA). * Gradle (included most of the time with the IDE).

Create a new plugin

Here are the steps:

1. Go on the plugin-template repository.

Structure

Figure 1: Structure

2. Click on *Use this template*.
3. Choose the GitHub account you want to link and the repository name for the new plugin.
4. Clone the new repository: `git clone git@github.com:{{user}}/{{name}}.git`.
5. Open the cloned directory in IntelliJ IDEA.
6. Enable annotations processors.
7. If you are using an IntelliJ IDEA $<$ 2020.03, install the lombok plugins (if not, it's included by default).

Once you completed the steps above, you should see a similar directory structure:

As you can see, there is one generated plugin: the `Example` class representing the `Example` plugin (a task).

A project typically hosts multiple plugins. We call a project a group of plugins, and you can have multiple sub-groups inside a project by splitting plugins into different packages. Each package that has a plugin class is a sub-group of plugins.

Plugin icons

Plugin icons need to be added in the SVG format — see an example here in the JIRA plugin.

Where can you find icons? - for proprietary systems, Wikipedia is a good source of SVG icons - for AWS services, the AWS icons is a great resource - Google Fonts Icons - Feather Icons.

Triggers

Manage Triggers in Kestra.

The **Triggers** page provides a concise overview of all triggers and their status, and allows you to disable, re-enable, or unlock triggers.

::alert{type="warning"} The API-side state of a trigger takes precedence over the state defined in the flow code. The state shown on this page is the authoritative source of truth for any trigger. Thus, if a trigger is marked as `disabled: true` in the source code but the UI toggle is on, the trigger is considered active despite being disabled in the code. ::

administration_triggers_ui

Figure 1: administration_triggers_ui

Flow Best Practices

How to design your workflows for optimal performance.

Understanding what is an execution internally for Kestra

The execution of a flow is an object that will contain: - All the TaskRuns for this flow, with each having: - Theirs attempts, with each having: - Theirs metrics - Their state histories - Theirs outputs - Their state histories

Internally: - Each TaskRun on a flow will be added on the same flow execution context that contains all tasks executed on this flow. - Each TaskRun status change is read by the Kestra Executor (at most 3 for a task: CREATED, RUNNING then SUCCESS). - For each state on the Executor, we need: - to fetch the serialized flow execution context over the network, - to deserialize the flow execution context, find the next task or tasks and serialize the flow execution context, - to send the serialized flow execution context over the network. - The bigger the flow execution context, the longer it will take to handle this serialization phase. - Depending on the Kestra internal queue and repository implementation, there can be a hard limit on the size of the flow execution context as it is stored as a single row/message. Usually, this limit is around 1MB, so this is important to avoid storing large amounts of data inside the flow execution context.

Task in the same execution

While it is possible to code a flow with any number of tasks, it is not recommended to have a lot of tasks on the same flow.

A flow can be comprised of manually generated or dynamic tasks. While EachSequential and EachParallel are really powerful tasks to loop over the result of a previous task, there are some drawbacks. If the task you are looping over is too large, you can easily end up with hundreds of tasks created. If, for example, you were using a pattern with `Each` inside `Each` (nested looping), it would take only a flow with 20 TaskRuns X 20 TaskRuns to reach 400 TaskRuns.

::alert{type="warning"} Based on our observations, we have seen that in cases where there are **more than 100** tasks on a flow, we see a decrease in performance and longer executions. ::

To avoid reaching these limits, you can easily create a subflow with the Subflow task, passing arguments from parent to child. In this case, since the `Subflow` task creates a new execution, the subflow tasks will be **isolated** and won't hurt performance.

Volume of data from your outputs

Some tasks allow you to fetch results on outputs to be reused on the next tasks. While this is powerful, this **is not intended to be used to transport a lot of data!** For example, with the Query task from BigQuery, there is a `fetch` property that allows fetching a result-set as an output attribute.

Imagine a big table with many megabytes or even gigabytes of data. If you use `fetch`, the output will be stored in the execution context and will need to be serialized on each task state change! This is not the idea behind `fetch`, it serves mostly to query a few rows to use it on a Switch task for example, or an EachParallel task to loop over.

::alert{type="info"} In most cases, there is a `stores` property that can handle a large volume of data. When an output is stored, it uses Kestra's internal storage, and only the URL of the stored file is stored in the execution context. ::

Parallel Task

Using the Parallel task is a convenient way to optimize flow duration, but keep in mind that, by default, **all parallel tasks are launched at the same time** (unless you specify the `concurrent` property). The only limit will be the number of worker threads you have configured.

Keep this in mind, because you cannot allow parallel tasks to reach the limit of external systems, such as connection limits or quotas.

Duration of Tasks

By default, Kestra **never limits the duration** (unless specified explicitly on the task's documentation) of the tasks. If you have a long-running process or an infinite loop, the tasks will never end. We can control the timeout on Runnable Tasks with the property timeout that takes a ISO 8601 duration like `PT5M` for a duration of 5 minutes.

Enterprise Edition (EE)

Learn about the Enterprise Edition and how it can help you run Kestra securely and reliably at scale.

Designed for production workloads with high security and compliance requirements, deployed in your private cloud.

Key Features

Kestra Enterprise is built on top of the Open Source Edition. However, it has a different architecture, with a few key differences listed below.

High Availability: Kestra Enterprise is designed to be highly-available and fault-tolerant. It uses a **Kafka** cluster as a backend for event-driven orchestration, and **Elasticsearch** for storing logs and metrics. This does not only improve performance, but also ensures that there is no single point of failure, and that the system can scale to handle large workloads.

Multi-Tenancy: the Enterprise Edition supports multi-tenancy, allowing you to create separate environments for different teams or projects. Each tenant is completely isolated from the others, and can be configured with its own access control policies.

Security and Access Control: Kestra Enterprise supports Single Sign-On (SSO) and Role-Based Access Control (RBAC), allowing you to integrate with your existing identity provider and manage user access to workflows and resources.

Enterprise Features: this Kestra edition comes with additional enterprise features, including Audit Logs, Worker Groups, Custom Blueprints, Namespace-level Secrets, Variables and Plugin Defaults.

Secrets Management: Kestra Enterprise can securely store and manage secrets. You can also integrate your existing secret manager, such as AWS Se-

login

Figure 1: login

cret Manager, Azure Key Vault, Elasticsearch, Google Secret Manager, and Hashicorp Vault.

Support: the Enterprise edition comes with guaranteed SLAs and priority support.

Onboarding: we provide onboarding and training for your team to help you get started quickly.

If you're interested to learn more, get in touch!

::alert{type="info"} **Kestra Cloud (Alpha):** if you don't have the capacity to host the Kestra Enterprise Edition yourself, you can try out Kestra Cloud, a fully managed SaaS product, hosted by the Kestra team. Kestra Cloud is currently in private Alpha. If you are interested in trying it out, sign up here. ::

Flow

Flow is a container for tasks and their orchestration logic.

Components of a flow

A flow is a container for `tasks`, their `inputs`, `outputs`, handling of `errors` and overall orchestration logic. It defines the **order** in which tasks are executed and **how** they are executed, e.g. **sequentially**, **in parallel**, based on upstream task dependencies and their state, etc.

You can define a flow declaratively using a YAML file.

A flow must have: - identifier (`id`) - `namespace` - list of `tasks`

Optionally, a flow can also have: - inputs - outputs - variables - triggers - labels - pluginDefaults - errors - retries - timeout - concurrency - descriptions - disabled - revision

Flow sample

Here is a sample flow definition. It uses tasks available in Kestra core for testing purposes, such as the `Return` or `Log` tasks, and demonstrates how to use `labels`, `inputs`, `variables`, `triggers` and various `descriptions`.

```yaml
id: hello-world
namespace: company.team

description: flow **documentation** in *Markdown*

labels:
  env: prod
  team: engineering

inputs:
  - id: my-value
    type: STRING
    required: false
    defaults: "default value"
    description: This is a not required my-value
```

enable disable flow

Figure 1: enable disable flow

```yaml
variables:
  first: "1"
  second: "{{vars.first}} > 2"

tasks:
  - id: date
    type: io.kestra.plugin.core.debug.Return
    description: "Some tasks **documentation** in *Markdown*"
    format: "A log line content with a contextual date variable {{taskrun.startDate}}"

pluginDefaults:
  - type: io.kestra.plugin.core.log.Log
    values:
      level: ERROR
```

Plugin defaults

You can also define `pluginDefaults` in your flow. This is a list of default task properties that will be applied to each task of a certain type inside your flow. The pluginDefaults property can be handy to avoid repeating the same values when leveraging the same task multiple times.

Variables

You can set flow variables that will be accessible by each task using `{{ vars.key }}`. Flow `variables` is a map of key/value pairs.

List of tasks

The most important part of a flow is the list of tasks that will be run sequentially when the flow is executed.

Disable a flow

By default, all flows are active and will execute whether or not a trigger has been set.

You have the option to disable a Flow, which is particularly useful when you want to temporarily stop a Flow from executing e.g. when troubleshooting a failure.

Task

A task is a single action in a flow. A task can have properties, use flow inputs and other task's outputs, perform an action, and produce an output.

There are two kinds of tasks in Kestra: - Runnable Tasks - Flowable Tasks

Runnable Task

Runnable Tasks handle computational work in the flow. For example, file system operations, API calls, database queries, etc. These tasks can be compute-intensive and are handled by workers.

By default, Kestra only includes a few Runnable Tasks. However, many of them are available as plugins, and if you use our default Docker image, plenty of them will already be included.

Flowable Task

Flowable Tasks only handle flow logic (branching, grouping, parallel processing, etc.) and start new tasks. For example, the Switch task decides the next task to run based on some inputs.

A Flowable Task is handled by an executor and can be called very often. Because of that, these tasks cannot include intensive computations, unlike Runnable Tasks. Most of the common Flowable Tasks are available in the default Kestra installation.

Labels

Labels are key-value pairs that you can add to flows. Labels are used to **organize** flows and can be used to **filter executions** of any given flow from the UI.

Inputs

Inputs are parameters sent to a flow at execution time. It's important to note that inputs in Kestra are strongly typed.

The inputs can be declared as either optional or mandatory. If the flow has required inputs, you'll have to provide them before the execution of the flow. You can also provide default values to the inputs.

Inputs can have validation rules that are enforced at execution time.

Inputs of type `FILE` will be uploaded to Kestra's internal storage and made available for all tasks.

Flow inputs can be seen in the **Overview** tab of the **Execution** page.

Outputs

Each task (or flow) can produce outputs that may contain multiple properties. This output is described in the plugin documentation task and can then be accessible by all following tasks via expressions.

Some outputs are of a special type and will be stored in Kestra's internal storage. Kestra will automatically make these outputs available for all tasks.

You can view: - task outputs in the **Outputs** tab of the **Execution** page. - flow outputs in the **Overview** tab of the **Execution** page.

If an output is a file from the internal storage, it will be available to download.

For more details on both task and flow outputs, see the Outputs page.

Revision

Changing the source of a flow will produce a new revision for the flow. The revision is an incremental number that will be updated each time you change the flow.

Internally, Kestra will track and manage all the revisions of the flow. Think of it as version control for your flows integrated inside Kestra.

You can access old revisions inside the **Revisions** tab of the **Flows** page.

Triggers

Triggers are a way to start a flow from external events. For example, a trigger might initiate a flow at a scheduled time or based on external events (webhooks, file creation, message in a broker, etc.).

Flow variable expressions

Flows have a number of variable expressions giving you information about them dynamically, a few examples include:

Parameter	Description
`{{ flow.id }}`	The identifier of the flow.
`{{ flow.namespace }}`	The name of the flow namespace.
`{{ flow.tenantId }}`	The identifier of the tenant (EE only).

Parameter	Description
`{{ flow.revision }}`	The revision of the flow.

Listeners (deprecated)

Listeners are special tasks that can listen to the current flow, and launch tasks *outside the flow*, meaning launch tasks that are not part of the flow.

The result of listeners will not change the execution status of the flow. Listeners are mainly used to send notifications or handle special behavior outside the primary flow.

Templates (deprecated)

Templates are lists of tasks that can be shared between flows. You can define a template and call it from other flows. Templates allow you to share a list of tasks and keep them updated without changing all flows that use them.

FAQ

Where does Kestra store flows?

Flows are stored in a serialized format directly **in the Kestra backend database**.

The easiest way to add new flows is to add them directly from the Kestra UI. You can also use the Git Sync pattern or CI/CD integration to add flows automatically after a pull request is merged to a given Git branch.

To see how flows are represented in a file structure, you can leverage the `_flows` directory in the Namespace Files editor.

How to load flows at server startup?

If you want to load a given local directory of flows to be loaded into Kestra (e.g. during local development), you can use the `-f` or `--flow-path` flag when starting Kestra:

```
./kestra server standalone -f /path/to/flows
```

That path should point to a directory containing YAML files with the flow definition. These files will be loaded to the Kestra repository at startup. Kestra will make sure to add flows to the right namespace, as declared in the flow YAML definition.

For more information about the Kestra server CLI, check the Server CLI Reference section.

Can I sync a local flows directory to be continuously loaded into Kestra?

At the time of writing, there is no syncing of a flows directory to Kestra. However, we are aware of that need and we are working on a solution. You can follow up in this GitHub issue.

Flows

Manage your flows in one place.

On the **Flows** page, you will see a list of flows which you can edit and execute. You can also create a new flow in the top right hand corner.

By clicking on a flow id or on the eye icon, you can open a flow.

A **Flow** page will have multiple tabs that allow you to: see the flow topology, all flow executions, edit the flow, view its revisions, logs, metrics, and dependencies. You'll also be able to edit namespace files in the Flow editor as well.

Editor

The Editor gives you a rich view of your workflow, as well as Namespace Files. The Editor allows you to add multipl views to the side: - Documentation - Topology - Blueprints

```
<iframe src="https://www.youtube.com/embed/o-d-GaXUiKQ?si=NR_-CYBsKhCqUNQ1" title="YouTube v
```

Topology View

The Topology View allows you to visualize the structure of your flow. This is especially useful when you have complex flows with multiple branches of logic.

Documentation View

The documentation view allows you to see Kestra's documentation right inside of the editor. As you move your type cursor around the Editor, the documentation page will update to reflect the specific task type documentation.

::alert{type="warning"} Note that if you use the Brave browser, you may need to disable the Brave Shields to make the Editor work as expected. Specifically, to view the task documentation, you need to set the `Block cookies` option to `Disabled` in the Shields settings: `brave://settings/shields`.

Kestra User Interface Flows Page

Figure 1: Kestra User Interface Flows Page

Kestra User Interface Flow Page

Figure 2: Kestra User Interface Flow Page

Topology

Figure 3: Topology

Brave cookies ::

Blueprints View

The blueprint view allows you to copy example flows directly into your flow. Especially useful if you're using a new plugin where you want to work off of an existing example.

Revisions

You can view the history of your flow under the Revisions tab. Read more about revisions here.

Dependencies

The Dependencies page allows you to view what other flows depend on the selected flow, as well as flows that the selected flow depends on. It gives you an easy way to navigate between them as well.

::alert{type="info"} The Dependencies View on the Namespaces page shows all the flows in the namespace and how they each relate to one another, if at all, whereas the Flow Dependencies view is only for the selected flow. ::

Docs

Figure 4: Docs

Blueprints Editor

Figure 5: Blueprints Editor

Blueprints Editor

Figure 6: Blueprints Editor

Dependencies

Figure 7: Dependencies

Fundamentals

Start by building a “Hello world” example.

::alert{type=“info”} To install Kestra, follow the Quickstart Guide or check the detailed Installation Guide. ::

Flows

Flows are defined in a declarative YAML syntax to keep the orchestration code portable and language-agnostic.

Each flow consists of three **required** components: `id`, `namespace` and `tasks`: 1. `id` represents the name of the flow 2. `namespace` can be used to separate development and production environments 3. `tasks` is a list of tasks that will be executed in the order they are defined

Here are those three components in a YAML file:

```yaml
id: getting_started
namespace: company.team
tasks:
  - id: hello_world
    type: io.kestra.plugin.core.log.Log
    message: Hello World!
```

The `id` of a flow must be **unique within a namespace**. For example: - you can have a flow named `getting_started` in the `company.team1` namespace and another flow named `getting_started` in the `company.team2` namespace. - you cannot have two flows named `getting_started` in the `company.team` namespace at the same time.

The combination of `id` and `namespace` serves as a **unique identifier** for a flow.

Namespaces

Namespaces are used to group flows and provide structure. Keep in mind that the allocation of a flow to a namespace is immutable. Once a flow is created, you cannot change its namespace. If you need to change the namespace of a flow, create a new flow with the desired namespace and delete the old flow.

Labels

To add another layer of organization, you can use labels, allowing you to group flows using key-value pairs.

Description(s)

You can optionally add a description property to keep your flows documented. The `description` is a string that supports **markdown** syntax. That markdown description will be rendered and displayed in the UI.

::alert{type="info"} Not only flows can have a description. You can also add a `description` property to `tasks` and `triggers` to keep all the components of your workflow documented. ::

Here is the same flow as before, but this time with **labels** and **descriptions**:

```yaml
id: getting_started
namespace: company.team

description: |
  # Getting Started
  Let's `write` some **markdown** - [first flow](https://t.ly/Vemr0)

labels:
  owner: rick.astley
  project: never-gonna-give-you-up

tasks:
  - id: hello_world
    type: io.kestra.plugin.core.log.Log
    message: Hello World!
    description: |
      ## About this task
      This task will print "Hello World!" to the logs.
```

Learn more about flows in the Flows section.

Tasks

Tasks are atomic actions in your flows. You can design your tasks to be small and granular, e.g. fetching data from a REST API or running a self-contained Python script. However, tasks can also represent large and complex processes, e.g. triggering containerized processes or long-running batch jobs (e.g. using dbt, Spark, AWS Batch, Azure Batch, etc.) and waiting for their completion.

Autocompletion

Figure 1: Autocompletion

The order of task execution

Tasks are defined in the form of a **list**. By default, all tasks in the list will be executed **sequentially** — the second task will start as soon as the first one finishes successfully.

Kestra provides additional **customization** allowing to run tasks **in parallel**, iterating (*sequentially or in parallel*) over a list of items, or to **allow failure** of specific tasks. Those are called `Flowable` tasks because they define the flow logic.

A task in Kestra must have an `id` and a `type`. Other properties depend on the task type. You can think of a task as a step in a flow that should execute a specific action, such as running a Python or Node.js script in a Docker container, or loading data from a database.

```yaml
tasks:
  - id: python
    type: io.kestra.plugin.scripts.python.Script
    containerImage: python:slim
    script: |
      print("Hello World!")
```

Autocompletion

Kestra supports hundreds of tasks integrating with various external systems. Use the shortcut `CTRL + SPACE` on Windows/Linux or `fn + control + SPACE` on Mac to trigger **autocompletion** listing available tasks or properties of a given task.

::alert{type="info"} If you want to **comment out** some part of your code, use the `CTRL or  + K + C` shortcut, and to uncomment it, use `CTRL or  + K + U`. To remember it, `C` stands for `comment` and `U` stands for `uncomment`. All available keyboard shortcuts are listed upon right-clicking anywhere in the code editor. ::

Supported task types

Let's look at supported task types.

Core

Core tasks from the `io.kestra.plugin.core.flow` category are commonly used to control the flow logic. You can use them to declare which processes should run **in parallel** or **sequentially**. You can specify **conditional branching**, **iterating** over a list of items, **pausing** or allowing certain tasks to fail without failing the execution.

Scripts

Script tasks are used to run scripts in Docker containers or local processes. You can use them to run Python, Node.js, R, Julia, or any other script. You can also use them to execute a series of commands in Shell or PowerShell. Check the Script tasks page for more details.

Internal Storage

Tasks from the `io.kestra.plugin.core.storage` category, along with Outputs, are used to interact with the **internal storage**. Kestra uses internal storage to **pass data between tasks**. You can think of internal storage as an S3 bucket. In fact, you can use your private S3 bucket as internal storage. This storage layer helps avoid proliferation of connectors. For example, you can use the Postgres plugin to extract data from a Postgres database and load it to the internal storage. Other task(s) can read that data from internal storage and load it to other systems such as Snowflake, BigQuery, or Redshift, or process it using any other plugin, without requiring point to point connections between each of them.

KV Store

Internal storage is mainly used to pass data within a single flow execution. If you need to pass data between different flow executions, you can use the **KV Store**. The tasks `Set`, `Get` and `Delete` from the `io.kestra.plugin.core.kv` category allow you to persist data between executions (even across namespaces). For example, if you are using dbt, you can leverage the KV Store to persist the `manifest.json` file between executions and implement the slim CI pattern.

Plugins

Apart from **core tasks**, the plugins library provides a wide range of integrations. Kestra has built-in plugins for data ingestion, data transformation, interacting with databases, object stores, or message queues, and the list keeps growing with every new release. On top of that, you can also create your own plugins to integrate with any system or programming language.

Create flow

Figure 2: Create flow

Create flow

Figure 3: Create flow

Create and run your first flow

Now, let's create and run your first flow. On the left side of the screen, click on the **Flows** menu. Then, click on the **Create** button.

Paste the following code to the Flow editor:

```yaml
id: getting_started
namespace: company.team

tasks:
  - id: api
    type: io.kestra.plugin.core.http.Request
    uri: https://dummyjson.com/products
```

Then, hit the **Save** button.

This flow has a single task that will fetch data from the dummyjson API. Let's run it!

::next-link Next, let's parametrize this flow using `inputs` ::

New execution

Figure 4: New execution

GitHub Actions

How to use GitHub Actions to create a CI/CD pipeline for your Kestra flows.

We provide two official GitHub Actions to help you create a CI/CD pipeline for your Kestra flows.

In GitHub Actions, CI/CD pipelines are called a Workflow, and is built with Actions performing validation and deployment of your flows.

To use the GitHub Actions, your Kestra installation must be accessible from the GitHub Actions runner. This means that your Kestra server must be accessible from the internet, or that you must use a self-hosted runner.

Kestra Actions

Kestra offers two Actions to create a CI/CD pipeline within a GitHub repository.

- Kestra Validate Action - Validate your flows and templates before deploying anything.
- Kestra Deploy Action - Deploy your flows and templates to your Kestra server.

Input Reference

Validate Inputs

Inputs	Required	Default	Description
`directory`	:heavy_check_mark:		Folder containing your resources
`resource`	:heavy_check_mark:		Resource you want to update in your namespace, can be `flow` or `template`
`server`	:x:		URL of your Kestra server, if none is provided, validation is done locally
`user`	:x:		User for the basic auth
`password`	:x:		Password for the basic auth
`apiToken`	:x:		API token for EE auth

Inputs	Required	Default	Description
`tenant`	:x:		Tenant identifier (EE only, when multi-tenancy is enabled)

Deploy Inputs

Inputs	Required	Default	Description
`namespace`	:heavy_check_mark:		Namespace containing your flows and templates
`directory`	:heavy_check_mark:		Folder containing your resources
`resource`	:heavy_check_mark:		Resource you want to update in your namespace, can be either `flow`,`template` or `namespace_files`
`server`	:heavy_check_mark:		URL of your Kestra server
`user`	:x:		User name of your Kestra server
`password`	:x:		Password of your Kestra server
`delete`	:x:	`true`	`Flows` found in Kestra server, but no longer existing in a specified directory, will be deleted by default. Set this to `false` if you want to avoid that behavior
`tenant`	:x:		Tenant identifier (EE only, when multi-tenancy is enabled)
`to`	:x:		Remote path indicating where to upload namespace files to

Examples

Here is an example of a Workflow using the Kestra actions to validate all Flows before deploying them.

```yaml
name: Kestra CI/CD
on: [push]

jobs:
  validate:
    runs-on: ubuntu-latest
    name: Kestra validate
    steps:
      - name: Checkout repo content
        uses: actions/checkout@v4

      - name: Validate all flows on server-side
        uses: kestra-io/validate-action@develop
        with:
          directory: ./kestra/flows
          resource: flow
```

```
        server: server_url

  # If validation passed, deploy resources
  deploy:
    runs-on: ubuntu-latest
    name: Kestra deploy
    steps:
      # We can only deploy to one namespace at once,
      # so we have two different steps for our two namespaces product and engineering
      - name: Checkout repo content
        uses: actions/checkout@v4

      - name: Deploy product flows
        uses: kestra-io/deploy-action@master
        with:
          namespace: product
          directory: ./kestra/flows/product
          resource: flow
          server: server_url

      - name: Deploy engineering flows
        uses: kestra-io/deploy-action@master
        with:
          namespace: engineering
          directory: ./kestra/flows/engineering
          resource: flow
          server: server_url
```

Check out the How-to Guide for more examples.

Gradle Configuration

We use Gradle as a build tool. This page will help you configure Gradle for your plugin.

Mandatory configuration

The first thing you need to configure is the plugin name and the class package.

1. Change in `settings.gradle` the `rootProject.name = 'plugin-template'` with your plugin name.
2. Change the class package: by default, the template provides a package `io.kestra.plugin.templates`, just rename the folder in `src/main/java` & `src/test/java`
3. Change the package name on `build.gradle`: replace `group "io.kestra.plugin.templates"` to the package name.

Now you can start developing your task or look at other optional gradle configuration.

Other configurations

Include some dependencies on plugins You can add as many dependencies to your plugins, they will be isolated in the Kestra runtime. Thanks to this isolation, we ensure that two different versions of the same library will not clash and have runtime errors about missing methods.

The `build.gradle` handle most of Kestra use case by default using `compileOnly group: "io.kestra", name: "core", version: kestraVersion` for Kestra libs.

But if your plugin need some dependencies, you can add as many as you want that will be isolated, you just need to add `api` dependencies:

```
api group: 'com.google.code.gson', name: 'gson', version: '2.8.6'
```

Main components

Technical description of Kestra's main components, including the internal storage, queue, repository, and plugins.

Kestra has three internal components: - The **Internal Storage** stores flow data like task outputs and flow inputs. - The **Queue** is used for internal communication between Kestra server components. - The **Repository** is used to store flows, templates, executions, logs, etc. The repository stores every internal object. - The **Plugins** extend the core of Kestra with new task and trigger types, storage implementations, etc.

These internal components are provided on multiple implementations depending on your needs and deployment architecture. You may need to install additional plugins to use some implementations.

Internal Storage

Kestra uses the concept of **Internal Storage** for storing input and output data. Multiple storage options are available, including local storage (default), S3 and Minio, Google Cloud Storage, and Azure Blobs Storage.

Check the Internal Storage documentation for more information.

Queue

The Queue, or more precisely, queues, are used internally for communication between the different Kestra server components. Kestra provides multiple queue types that must be used with their repository counterparts.

There are three types of queues: - **In-Memory** that must be used with the In-Memory Repository. - **Database** that must be used with the Database Repository. - **Kafka** that must be used with the Elasticsearch Repository. **Only available in the Enterprise Edition**.

Repository

The Repository, or more precisely, repositories, are the internal way to store data. Kestra provides multiple repository types that must be used with their queue counterparts.

There exist three types of repositories: - **In-Memory** that must be used with the In-Memory Queue. - **Database** that must be used with the Database Queue. - **Elasticsearch** that must be used with the Kafka Queue. **Only available in the Enterprise Edition**.

Plugins

Kestra's core is not able to handle a lot of task types on its own. We have therefore, included a Plugins' ecosystem that allows developing as many task types as you need. A wide range of plugins are already available, and many more will be delivered by the Kestra team!

Plugins are also used to provide different implementations for Kestra's internal components like its Internal Storage.

Task Runner Overview

Task Runner capabilities and supported plugins.

Task Runners Capabilities

Task Runners offer a powerful way to offload compute-intensive tasks to remote environments. The table below highlights their capabilities.

Capability	Description
Fine-grained resource allocation	Task Runners give you full control over the compute resources — you can flexibly choose how much CPU, memory, or GPU you want to allocate to specific tasks.
Flexible deployment patterns	Task Runners support various deployment models, including AWS ECS Fargate, Azure Batch, Google Batch, Kubernetes, and more. You can mix and match different runners even within a single workflow.
No vendor lock-in	Task Runners are built on top of a plugin ecosystem, so you can run your code on any cloud provider or on-premises infrastructure without being locked to a specific vendor or deployment model.
Task isolation	Your tasks run in fully isolated container environments without interfering with each other or competing for resources.
Made for development and production	You can develop your code locally in Docker containers and run the same code in a production environment on a Kubernetes cluster. Thanks to task runners, setting this up is as simple as changing a single property.
Centralized configuration management	Task Runners make it easy to centrally govern your configuration. For example, you can use `pluginDefaults` on a namespace level to manage your task runner configuration and credentials in a single place.

Capability	Description
Documentation and autocompletion	Each task runner is a plugin with its own schema. The built-in code editor provides documentation, autocompletion and syntax validation for all runner properties to ensure correctness, standardization and consistency.
No changes to your code	You can run the same business logic in different environments without changing anything in your code.
Fully customizable	If you need more customization, you can create your own Task Runner plugin to match your specific deployment patterns.

Plugins Supporting Task Runners

Task Runners are intended to be used in the tasks from the Script Plugin and its sub-plugins tasks, including: - Python - Node - Shell - PowerShell - R - Julia - Ruby - dbt - Singer - SQLMesh - Ansible - Terraform - Modal - AWS CLI - GCloud CLI - Azure CLI

Anytime you see a task that can execute a `script` or a series of `commands`, it's a script task that contains a `taskRunner` property.

Process Task Runner

Run tasks as local processes.

How to use the Process task runner

Here is an example of a Shell script configured with the Process task runner which runs a Shell command as a child process in the Kestra host:

```yaml
id: process_script_runner
namespace: company.team

tasks:
  - id: shell
    type: io.kestra.plugin.scripts.shell.Commands
    taskRunner:
      type: io.kestra.plugin.core.runner.Process
    commands:
      - echo "Hello World!"
```

The Process task runner doesn't have any additional configuration beyond the `type` property.

Benefits

The Process task runner is useful if you want to access local files e.g. to take advantage of locally configured software libraries and virtual environments.

Combining task runners with Worker Groups

You can combine the Process task runner with Worker Groups to run tasks on dedicated servers that might have specific software libraries or configurations. This powerful combination allows you to leverage the compute resources of your Worker Groups while running tasks as local processes without the overhead of containerization.

The example below shows how to combine the Process task runner with Worker Groups to fully leverage the GPU resources of a dedicated server:

```yaml
id: python_on_gpu
namespace: company.team

tasks:
  - id: gpu_intensive_ai_workload
    type: io.kestra.plugin.scripts.python.Commands
    namespaceFiles:
      enabled: true
    commands:
      - python main.py
    workerGroup:
      key: gpu
    taskRunner:
      type: io.kestra.plugin.core.runner.Process
```

::alert{type="info"} Note that Worker Group is an Enterprise Edition functionality. If you want to try it out, please reach out. ::

Quickstart

Start Kestra in a Docker container and create your first flow.

Start Kestra

::alert{type="info"} **Prerequisites**: Make sure that Docker is installed in your environment. We recommend Docker Desktop. :: Make sure that Docker is running. Then, you can start Kestra in a single command using Docker (*if you run it on Windows, make sure to use WSL*):

```
docker run --pull=always --rm -it -p 8080:8080 --user=root -v /var/run/docker.sock:/var/run/
```

Open `http://localhost:8080` in your browser to launch the UI and start building your first flows.

::alert{type="info"} The above command starts Kestra with an embedded H2 database that will not persist data. If you want to use a persistent database backend with Postgres and more configurability, follow the Docker Compose installation. ::

Create Your First Flow

Navigate to **Flows** in the left menu, then click the "Create" button and paste the following configuration to create your first flow:

```yaml
id: getting_started
namespace: company.team
tasks:
  - id: hello_world
    type: io.kestra.plugin.core.log.Log
    message: Hello World!
```

Click on **Save** and then on the **Execute** button to start your first execution.

::next-link For a more detailed introduction to Kestra, check our Tutorial ::

Next Steps

Congrats! You've just installed Kestra and executed your first flow! :clap:

Next, you can follow the documentation in this order: - Check out the tutorial - Get to know the building blocks of a flow - Learn the core concepts - Check out the available Plugins to integrate with external systems and start orchestrating your applications, microservices and processes - Deploy Kestra to remote development and production environments - Almost everything is configurable in Kestra. You can find the different configuration options available to Administrators in the Configuration Guide

Runnable Tasks

Data processing tasks handled by the workers.

Runnable tasks are data processing tasks incl. file system operations, API calls, database queries, etc. These tasks can be compute-intensive and are processed by workers.

Each task must have an identifier (id) and a type. The type is the task's Java Fully Qualified Class Name (FQCN).

Tasks have properties specific to the type of the task; check each task's documentation for the list of available properties.

Most available tasks are Runnable Tasks except special ones that are Flowable Tasks.

By default, Kestra only includes a few Runnable Tasks. However, many of them are available as plugins, and if you use our default Docker image, plenty of them will already be included.

Example

In this example, we have 2 Runnable Tasks: one which makes a HTTP request and another that logs the output of that request.

```yaml
id: runnable_http
namespace: company.team

tasks:
  - id: make_request
    type: io.kestra.plugin.core.http.Request
    uri: https://reqres.in/api/products
    method: GET
    contentType: application/json

  - id: print_status
    type: io.kestra.plugin.core.log.Log
    message: "{{ outputs.make_request.body }}"
```

Schedule Trigger

Schedule flows with cron expressions.

The Schedule trigger generates new executions on a regular cadence based on a Cron expression or custom scheduling conditions.

```
type: "io.kestra.plugin.core.trigger.Schedule"
```

Kestra is able to trigger flows based on a Schedule (aka the time). If you need to wait for another system to be ready and cannot use any event mechanism, you can schedule one or more time the current flow.

Kestra will optionally handle schedule backfills if any executions are missed.

Check the Schedule task documentation for the list of the task properties and outputs.

Example: A schedule that runs every quarter of an hour.

```
triggers:
  - id: schedule
    type: io.kestra.plugin.core.trigger.Schedule
    cron: "*/15 * * * *"
```

A schedule that runs only the first monday of every month at 11 AM.

```
triggers:
  - id: schedule
    type: io.kestra.plugin.core.trigger.Schedule
    cron: "0 11 * * 1"
    conditions:
      - type: io.kestra.plugin.core.condition.DayWeekInMonthCondition
        date: "{{ trigger.date }}"
        dayOfWeek: "MONDAY"
        dayInMonth: "FIRST"
```

A schedule that runs daily at midnight US Eastern time.

```
triggers:
  - id: daily
    type: io.kestra.plugin.core.trigger.Schedule
```

```
    cron: "@daily"
    timezone: America/New_York
```

::alert{type="warning"} Schedules **cannot overlap**. This means that we **cannot have any concurrent schedules**. If the previous schedule is not ended when the next one must start, the scheduler will wait until the end of the previous one. The same applies during backfills. ::

::alert{type="info"} Most of the time, schedule execution will depend on the `trigger.date` (looking at files for today, SQL query with the schedule date in the where clause, …). This works well but prevents you from executing your flow manually (since these variables are only available during the schedule).

You can use this expression to make your **manual execution work**: `{{ trigger.date ?? execution.startDate | date("yyyy-MM-dd") }}`. It will use the current date if there is no schedule date making it possible to start the flow manually. ::

Schedule Conditions

When the `cron` is not sufficient to determine the date you want to schedule your flow, you can use `conditions` to add additional conditions, (for example, only the first day of the month, only the weekend, …).

You **must** use the `{{ trigger.date }}` expression on the property `date` of the current schedule.

This condition will be evaluated and `{{ trigger.previous }}` and `{{ trigger.next }}` will reflect the date **with** the conditions applied.

The list of core conditions that can be used are:

- DateTimeBetweenCondition
- DayWeekCondition
- DayWeekInMonthCondition
- NotCondition
- OrCondition
- WeekendCondition
- PublicHolidayCondition
- TimeBetweenCondition

Here's an example using the `DayWeekCondition`:

```
id: conditions
namespace: company.team

tasks:
  - id: hello
    type: io.kestra.plugin.core.log.Log
    message: This will execute only on Thursday!
```

```yaml
triggers:
  - id: schedule
    type: io.kestra.plugin.core.trigger.Schedule
    cron: "@hourly"
    conditions:
      - type: io.kestra.plugin.core.condition.DayWeekCondition
        dayOfWeek: "THURSDAY"
```

Recover Missed Schedules

Automatically

If a schedule is missed, Kestra will automatically recover it by default. This means that if the Kestra server is down, the missed schedules will be executed as soon as the server is back up. However, this behavior is not always desirable, e.g. during a planned maintenance window. In Kestra 0.15 and higher, this behavior can be disabled by setting the `recoverMissedSchedules` configuration to `NONE`.

Kestra 0.15 introduced a new configuration allowing you to choose whether you want to recover missed schedules or not:

```yaml
kestra:
  plugins:
    configurations:
      - type: io.kestra.plugin.core.trigger.Schedule
        values:
          # available options: LAST | NONE | ALL -- default: ALL
          recoverMissedSchedules: NONE
```

The `recoverMissedSchedules` configuration can be set to `ALL`, `NONE` or `LAST`: - `ALL`: Kestra will recover all missed schedules. This is the **default** value. - `NONE`: Kestra will not recover any missed schedules. - `LAST`: Kestra will recover only the last missed schedule for each flow.

Note that this is a global configuration that will apply to all flows, unless other behavior is explicitly defined within the flow definition:

```yaml
triggers:
  - id: schedule
    type: io.kestra.plugin.core.trigger.Schedule
    cron: "*/15 * * * *"
    recoverMissedSchedules: NONE
```

In this example, the `recoverMissedSchedules` is set to `NONE`, which means that Kestra will not recover any missed schedules for this specific flow regardless of the global configuration.

backfill1

Figure 1: backfill1

Using Backfill

Backfills are replays of missed schedule intervals between a defined start and end date.

To backfill the missed executions, go to the `Triggers` tab on the Flow's detail page and click on the `Backfill executions` button.

For more information on Backfill, check out the dedicated documentation.

Disabling the trigger If you're unsure what how you want to proceed and need to time to decide, you can disable the trigger by either adding the `disabled: true` property to your YAML or by toggling on the Triggers page.

This is useful if you're figuring out what to do before the next schedule is due to run.

For more information on Disabled, check out the dedicated documentation.

Setting Inputs inside of the Schedule trigger

You can easily pass inputs to the Schedule Trigger by using the `inputs` property and passing them as a key-value pair.

In this example, the `user` input is set to "John Smith" inside of the `schedule` trigger:

```yaml
id: myflow
namespace: company.team

inputs:
  - id: user
    type: STRING
    defaults: Rick Astley

tasks:
  - id: hello
    type: io.kestra.plugin.core.log.Log
    message: "Hello {{ inputs.user }}!  "

triggers:
  - id: schedule
    type: io.kestra.plugin.core.trigger.Schedule
    cron: "*/1 * * * *"
```

```yaml
inputs:
  user: John Smith
```

Workers

Manage Workers in Kestra.

On the **Cluster** page, navigate to **Services** tab. You will see the list of available workers by filtering `Worker` from the **Type** dropdown.

Kestra User Interface Workers

Figure 1: Kestra User Interface Workers

Moving from Development to Production

Common patterns to deploy your flows from development to production environments.

Development Environment

One best practice with Kestra is to have one development instance where users can write their flow directly in UI. This instance can be seen as a "sandbox" where flows can be tested and executed without the fear to break critical business operations.

We usually encourage two types of development environment: - installing Kestra on your local machine (usually with Docker Compose installation) - installing Kestra on a Kubernetes cluster accessible by users and separated from production matters.

Production Environment

The production instance should be safeguarded. Especially as this environment supports critical operations and engages your responsibilities for end users.

One common best practice here is to limit the access of the production environment. In this case, there two elements to consider: - User access - Flow deployments

User Access

For Kestra Enterprise users, this is streamlined with RBAC and SSO features. With role policies such as "Admin" or "Viewer", one administrator can manage all user access with fine-grain control over all Kestra resources. You can learn more in the dedicated documentation.

For open-source users it's usually a good idea to have a restricted instance, meaning an instance only accessible by CI/CD and administrators.

Flows Deployment

Kestra offers many strategies to deploy flows to an instance: - Through the UI - Sync with Git - CI/CD - Terraform - API.

Choosing one way or the other depends of your preferences and your current deployment patterns.

One recurring pattern is moving flows from the development to the production instance through version control system and CI/CD.

When users have developed flows, they will usually commit changes to a version control system (Git). Then, upon validated pull request, the CI/CD engine will deploy the corresponding flows to the production instance.

The way users can commit flow changes to Git can be addressed with the following patterns: - Export or copy-paste flows from the user interface - Using the `git.PushFlows` task

The way CI/CD deploy flows to production instance can be addressed with the following patterns: - GitHub Action, GitLab CI/CD, Jenkins, Azure DevOps, etc. - Terraform deployment - Kestra CLI

You can find more about CI/CD pattern with Kestra here.

Git Example

```
<iframe src="https://www.youtube.com/embed/02bFAu-rpxU?si=bzj_Gs_mxxocdhd2" title="YouTube v
```

We can use the `git.SyncFlows` task combined with a Trigger to automatically pull Flows from the `main` branch of the Git repository.

This means Kestra will manage the process, reducing the number of systems needed to automate the dev to prod process.

You can use either a Schedule Trigger to pull on a regular routine, e.g. nightly, or the Webhook Trigger to pull when the `main` branch receives new commits. Check out this dedicated guide on setting them up.

If we combine this with the `git.PushFlows` task, we can push our Flows to our repository from our local development environment, ensuring they are validated as Kestra will not let you save an invalid flow.

On top of that, we can automatically open a Pull Request with the `create.Pulls` task to `main` to speed up the process of getting the Flows to production.

::alert{type="info"} While we can be sure that our flows are valid, this will not check for logical errors. We'd recommend testing flows separately to check this before deploying to production. ::

CI/CD Example

We can use CI/CD to automatically deploy our flows from our Git repository to our production instance of Kestra when they are merged to the `main` branch.

With GitHub, we can use the official Deploy Action, which uses the Kestra Server CLI under the hood, to deploy when a Pull Request is merged to `main`.

We can combine the **Deploy Action** with the Validate Action, which runs a validate check on Flows using the Kestra Server CLI.

This means we can configure the Git Repository to require status checks to pass before a Pull Request can be merged - preventing any invalid flows from being deployed to production.

::alert{type="info"} **Note:** If a flow is in an invalid format, the **Deploy Action** will fail. ::

Task Runner Benefits

How Task Runners can help with resource allocation and environment management.

Docker in development, Kubernetes in production

Many Kestra users develop their scripts locally in Docker containers and then run the same code in a production environment as Kubernetes pods. Thanks to the `taskRunner` property, setting this up is a breeze. Below is an example showing how you can combine `pluginDefaults` with the `taskRunner` property to use Docker in the development environment and Kubernetes in production — all without changing anything in your code.

1. Development namespace/tenant/instance:

```yaml
pluginDefaults:
  - type: io.kestra.plugin.scripts
    values:
      taskRunner:
        type: io.kestra.plugin.scripts.runner.docker.Docker
        pullPolicy: IF_NOT_PRESENT # in dev, only pull the image when needed
        cpu:
          cpus: 1
        memory:
          memory: 512Mi
```

2. Production namespace/tenant/instance:

```yaml
pluginDefaults:
  - type: io.kestra.plugin.scripts
    values:
      taskRunner:
        type: io.kestra.plugin.ee.kubernetes.runner.Kubernetes
        namespace: company.team
        pullPolicy: ALWAYS # Always pull the latest image in production
        config:
          username: "{{ secret('K8S_USERNAME') }}"
          masterUrl: "{{ secret('K8S_MASTER_URL') }}"
          caCert: "{{ secret('K8S_CA_CERT') }}"
```

docker_runner

Figure 1: docker_runner

```
      clientCert: "{{ secret('K8S_CLIENT_CERT') }}"
      clientKey: "{{ secret('K8S_CLIENT_KEY') }}"
    resources: # can be overriden by a specific task if needed
      request: # The resources the container is guaranteed to get
        cpu: "500m" # Request 1/2 of a CPU (500 milliCPU)
        memory: "256Mi" # Request 256 MB of memory
```

::alert{type="info"} Note how the `containerImage` property is not included in the `taskRunner` configuration but as a generic property available to any scripting task. This makes the configuration more flexible as usually the container image changes more often than the standard runner configuration. For instance, the dbt plugin may need a different image than the generic Python plugin, but the runner configuration can stay the same. ::

Centralized configuration management

The combination of `pluginDefaults` and `taskRunner` properties allows you to centrally manage your task runner configuration. For example, you can use `pluginDefaults` on a namespace level to centrally manage your AWS credentials for the `Batch` task runner plugin.

```
pluginDefaults:
  - type: io.kestra.plugin.ee.aws.runner.Batch
    values:
      accessKeyId: "{{ secret('AWS_ACCESS_KEY_ID') }}"
      secretKeyId: "{{ secret('AWS_SECRET_ACCESS_KEY') }}"
      region: "us-east-1"
```

Documentation and autocompletion

Each task runner is a plugin with its own icon, documentation, and schema to validate its properties. The built-in code editor in the Kestra UI provides autocompletion and syntax validation for all runner properties, and when you click on the runner's name in the editor, you can see its documentation on the right side of the screen.

Full customization: create your own Task Runner

You can create a custom task runner plugin for your specific environment, build it as a JAR file, and add that file to the `plugins` directory. Once you restart Kestra, your custom runner plugin will be available on any script task in the system.

Docker

Start Kestra in a single Docker container.

Make sure that Docker is running. Then, you can start Kestra in a single command using Docker (*if you run it on Windows, make sure to use WSL*):

```
docker run --pull=always --rm -it -p 8080:8080 --user=root -v /var/run/docker.sock:/var/run/
```

Open http://localhost:8080 in your browser to launch the UI and start building your first flow.

::alert{type="info"} The above command starts Kestra with an embedded H2 database. If you want to use a persistent database backend with PostgreSQL and more configurability, follow the Docker Compose installation. ::

Configuration

Using a configuration file

You can adjust Kestra's configuration using a file passed to the Docker container as a bind volume.

First, create a configuration file, for example, in a file named `application.yaml`:

```yaml
datasources:
  postgres:
    url: jdbc:postgresql://postgres:5432/kestra
    driverClassName: org.postgresql.Driver
    username: kestra
    password: k3str4
kestra:
  server:
    basicAuth:
      enabled: false
      username: "admin@kestra.io" # it must be a valid email address
      password: kestra
  repository:
    type: postgres
  storage:
```

```
    type: local
    local:
      basePath: "/app/storage"
  queue:
    type: postgres
  tasks:
    tmpDir:
      path: "/tmp/kestra-wd/tmp"
  url: "http://localhost:8080/"
```

::alert{type="info"} Note: this configuration is taken from our official docker-compose.yaml file and uses a PostgreSQL database, you may want to retrieve it there to be sure it is accurate. ::

Then, change the command to mount the file to the container and start Kestra with the configuration file; we also adjust the Kestra command to start a standalone version as we now have a PostgreSQL database as a backend.

```
docker run --pull=always --rm -it -p 8080:8080 --user=root \
 -v $PWD/application.yaml:/etc/config/application.yaml \
 -v /var/run/docker.sock:/var/run/docker.sock \
 -v /tmp:/tmp kestra/kestra:latest server standalone --config /etc/config/application.yaml
```

Using the `KESTRA_CONFIGURATION` environment variable

You can adjust the Kestra configuration with a `KESTRA_CONFIGURATION` passed to the Docker container via the `-e` options. This environment variable must be a valid YAML string.

Managing a large configuration via a single YAML string can be tedious. To make that easier, you can leverage a configuration file.

First, define an environment variable:

```
export KESTRA_CONFIGURATION="datasources:
          postgres:
            url: jdbc:postgresql://postgres:5432/kestra
            driverClassName: org.postgresql.Driver
            username: kestra
            password: k3str4
        kestra:
          server:
            basicAuth:
              enabled: false
              username: "admin@kestra.io" # it must be a valid email address
              password: kestra
          repository:
            type: postgres
          storage:
```

```
  type: local
  local:
    basePath: "/app/storage"
queue:
  type: postgres
tasks:
  tmpDir:
    path: /tmp/kestra-wd/tmp
url: http://localhost:8080/
```

::alert{type="info"} Note: this configuration is taken from our official docker-compose.yaml file and uses a PostgreSQL database, you may want to retrieve it there to be sure it is accurate. ::

Then pass this environment variable in the Docker command and adjust the Kestra command to run the standalone server:

```
docker run --pull=always --rm -it -p 8080:8080 --user=root \
 -e KESTRA_CONFIGURATION=$KESTRA_CONFIGURATION
 -v /var/run/docker.sock:/var/run/docker.sock \
 -v /tmp:/tmp kestra/kestra:latest server standalone
```

Official Docker images

The official Kestra Docker images are available on DockerHub for both `linux/amd64` and `linux/arm64` platforms.

We provide two image variants: - `kestra/kestra:*` - `kestra/kestra:*-no-plugins`

Both variants are based on the `eclipse-temurin:21-jre` Docker image.

The `kestra/kestra:*` images contain all Kestra plugins in their **latest version**. The `kestra/kestra:*-no-plugins` images do not contain any plugins. We recommend using the `kestra/kestra:*` version.

Docker image tags

We provide the following tags for each Docker image:

- `latest`: the default image with the latest stable release, including all plugins.
- `latest-no-plugins`: the default image with the latest stable release, excluding all plugins.
- `v<release-version>`: image for a specific Kestra release, including all plugins.
- `v<release-version>-no-plugins`: image for a specific Kestra release, excluding all plugins.
- `develop`: an image based on the `develop` branch that changes daily and contains **unstable** features we are working on, including all plugins.

- `develop-no-plugins`: an image based on the `develop` branch that changes daily and contains **unstable** features we are working on, excluding all plugins.

The **default Kestra image** `kestra/kestra:latest` already includes **all plugins**. To use a lightweight version of Kestra without plugins, add a suffix `*-no-plugins`.

Recommended images for production

We recommend using the `latest` image for production deployments. This image includes the latest stable release and optionally also all plugins: - `kestra/kestra:latest` — contains the latest stable version of Kestra and all plugins - `kestra/kestra:latest-no-plugins` — contains the latest stable version of Kestra without any plugins.

If your deployment strategy is to pin the version, make sure to change the image as follows (*here, based on the `v0.18.0` release*): - `kestra/kestra:v0.18.0` if you want to have all plugins included in the image - `kestra/kestra:v0.18.0-no-plugins` if you prefer to use only your custom plugins.

Recommended images for development

The most recently developed (but not yet released) features and bug fixes are included in the `develop` image. This image is updated daily and contains the latest changes from the `develop` branch: - `kestra/kestra:develop` if you want to have all plugins included in the image - `kestra/kestra:develop-no-plugins` if you prefer to use only your custom plugins.

Build a custom Docker image

If the base or full image doesn't contain package dependencies you need, you can build a custom image by using the Kestra base image and adding the required binaries and dependencies.

Add custom binaries

The following `Dockerfile` creates an image from the Kestra base image and adds the `golang` binary and Python packages:

```
ARG IMAGE_TAG=latest
FROM kestra/kestra:$IMAGE_TAG

RUN mkdir -p /app/plugins && \
  apt-get update -y && \
  apt-get install -y --no-install-recommends golang && \
  apt-get install -y pip && \
```

```
  pip install pandas==2.0.3 requests==2.31.0 && \
  apt-get clean && rm -rf /var/lib/apt/lists/* /var/tmp/*
```

Add plugins to a Docker image

By default, the base Docker image `kestra/kestra:latest` contains all plugins (unless you use the `kestra/kestra:latest-no-plugins` version). You can add specific plugins to the base image and build a custom image.

The following example `Dockerfile` creates an image from the base image and adds the `plugin-notifications`, `storage-gcs` and `plugin-gcp` binaries using the command `kestra plugins install`:

```
ARG IMAGE_TAG=latest-no-plugins
FROM kestra/kestra:$IMAGE_TAG

RUN /app/kestra plugins install \
  io.kestra.plugin:plugin-notifications:LATEST \
  io.kestra.storage:storage-gcs:LATEST \
  io.kestra.plugin:plugin-gcp:LATEST
```

Add custom plugins to a Docker image

The above example `Dockerfile` installs plugins that have already been published to Maven Central. If you are developing a custom plugin, make sure to build it. Once the `shadowJar` is built, add it to the plugins directory:

```
ARG IMAGE_TAG=latest
FROM kestra/kestra:$IMAGE_TAG

RUN mkdir -p /app/plugins

COPY /build/libs/*.jar /app/plugins
```

Add custom plugins from a Git repository

If you would like to build custom plugins from a specific Git repository, you can use the following approach:

```
FROM openjdk:17-slim as stage-build
WORKDIR /
USER root

RUN apt-get update -y
RUN apt-get install git -y && \
    git clone https://github.com/kestra-io/plugin-aws.git

RUN cd plugin-aws && ./gradlew :shadowJar
```

```
FROM kestra/kestra:latest

# https://github.com/WASdev/ci.docker/issues/194#issuecomment-433519379
USER root

RUN mkdir -p /app/plugins && \
  apt-get update -y && \
  apt-get install -y --no-install-recommends golang && \
  apt-get install -y pip && \
  pip install pandas==2.0.3 requests==2.31.0 && \
  apt-get clean && rm -rf /var/lib/apt/lists/* /var/tmp/*

RUN rm -rf /app/plugins/plugin-aws-*.jar
COPY --from=stage-build /plugin-aws/build/libs/plugin-aws-*.jar /app/plugins
```

This multi-stage Docker build allows you to overwrite a plugin that has already been installed. In this example, the AWS plugin is by default already included in the `kestra/kestra:latest` image. However, it's overwritten by a plugin built in the first Docker build stage.

Docker Task Runner

Run tasks as Docker containers.

How to use the Docker task runner

Here is an example using the Docker task runner executing the commands in a Docker container:

```yaml
id: docker_script_runner
namespace: company.team

tasks:
  - id: shell
    type: io.kestra.plugin.scripts.shell.Commands
    containerImage: centos
    taskRunner:
      type: io.kestra.plugin.scripts.runner.docker.Docker
      cpu:
        cpus: 1
    commands:
      - echo "Hello World!"
```

Once you specify the `taskRunner` type, you get the autocompletion and validation for the runner-specific properties. In the example above, the task allocates 1 CPU to the container.

Docker task runner properties

The only property required by the `taskRunner` is the `containerImage` property that needs to be set on the script task. The image can be from a public or private registry.

Additionally, using the Docker task runner you can configure memory allocation, volumes, environment variables, and more. For a full list of properties available

docker_runner

Figure 1: docker_runner

in the Docker task runner, check the Docker plugin documentation or explore the same in the built-in Code Editor in the Kestra UI.

::alert{type="info"} The Docker task runner executes the script task as a container in a Docker-compatible engine. This means that you can use it to run scripts within a Kubernetes cluster with Docker-In-Docker (dind) or in a local Docker engine as well. ::

Task runner behavior in a failure scenario

Generally speaking, each task runner container initiated by Kestra will **continue running until the task completes**, even if the Kestra worker is terminated (e.g. due to a crash). However, there are some caveats to be aware of depending on how Kestra and the task runner are deployed.

Kestra running in a Docker container, Task Runner running in DinD

When Kestra runs in a Docker container and uses DinD for task runners, terminating the Kestra container will also terminate the DinD container and any running task containers inside DinD. No container is automatically restarted.

Kestra running in Kubernetes, Task Runner running in DinD

When Kestra and DinD are deployed in the same pod in a Kubernetes environment, the pod will be restarted if Kestra Worker fails. This ensures that the DinD container and any task runner containers are also restarted.

Kestra deployed with Docker-Compose, Task Runner running in DinD

When using Docker-Compose, Kestra and DinD containers can be managed independently. Restarting the Kestra container does NOT automatically restart the DinD container. Therefore, task runners running inside DinD may continue running even if Kestra is restarted.

Insecure Registry

The Docker task runner supports insecure registries. Prior to using this feature, ensure the insecure registry is configured on the host machine that your Kestra server is running on.

For example, if you have the insecure registry `10.10.1.5:5000`, please add the following configuration to `/etc/docker/daemon.json` and then restart your Docker daemon.

```
{
  "insecure-registries" : ["10.10.1.5:5000"]
}
```

This can then be used in a flow with the following YAML

```yaml
id: docker_example
namespace: demo

tasks:
  - id: my_command
    type: io.kestra.plugin.scripts.shell.Commands
    taskRunner:
      type: io.kestra.plugin.scripts.runner.docker.Docker
      config: |
        {
          "insecure-registries" : ["10.10.1.5:5000"]
        }
    containerImage: 10.10.1.5:5000/my-image
    commands:
      - echo "Hello World!"
```

Executions

Manage Executions of your Flows in one place.

On the **Executions** page, you will see a list of previous flow executions.

By clicking on an execution id or on the eye icon, you can open an execution.

An **Execution** page will allow you to access the details of a flow execution, including logs, outputs, and metrics.

Gantt

The Gantt tab allows you to see each task's durations. From this interface, you can replay a specific task, see task source code, change task status, or look at task metrics and outputs.

Logs

The Logs tab allows to access task's logs. You can filter by log level, copy logs in your clipboard, or download logs as a file.

Topology

Similar to the Editor view, you can see your execution topology. From this, you can access specific task logs, replay certain tasks or change task status.

Outputs

The Outputs tab inside of an execution page allows to see each task's outputs.

The "Debug Outputs" box allows to evaluate expressions on those task outputs. It's a great way to debug your flows.

> Note: You have to select one task to be able to use the "Debug Outputs" button.

Kestra User Interface Executions Page

Figure 1: Kestra User Interface Executions Page

Kestra User Interface Execution Page

Figure 2: Kestra User Interface Execution Page

Kestra User Interface Execution Gantt

Figure 3: Kestra User Interface Execution Gantt

For example, you can use the “Render expression” feature to deep-dive into your tasks’ outputs and play directly with expressions.

Metrics

The Metrics tab shows every metric exposed by tasks after execution.

Kestra User Interface Execution Log

Figure 4: Kestra User Interface Execution Log

Kestra User Interface Execution Topology

Figure 5: Kestra User Interface Execution Topology

Kestra User Interface Execution Outputs

Figure 6: Kestra User Interface Execution Outputs

Kestra User Interface Execution Debug Outputs

Figure 7: Kestra User Interface Execution Debug Outputs

Kestra User Interface Execution Metric

Figure 8: Kestra User Interface Execution Metric

Flow Trigger

Trigger flows from another flow execution.

Flow triggers allows you to trigger a flow after another flow execution, enabling event-driven patterns.

```
type: "io.kestra.plugin.core.trigger.Flow"
```

Kestra is able to trigger one flow after another one. This allows the chaining of flows without the need to update the base flows. With this capacity, you can break responsibility between different flows to different teams.

Check the Flow trigger documentation for the list of all properties.

Conditions

You can provide conditions to determine when your Flow should be executed. Along with the core trigger conditions, you can use the following:

- ExecutionFlowCondition
- ExecutionNamespaceCondition
- ExecutionLabelsCondition
- ExecutionStatusCondition
- ExecutionOutputsCondition
- ExpressionCondition

Example

This flow will be triggered after each successful execution of the flow `io.kestra.tests.trigger-flow` and forward the `uri` output of the `my-task` task.

```
id: trigger_flow_listener
namespace: company.team

inputs:
  - id: fromParent
    type: STRING

tasks:
```

```
  - id: onlyNoInput
    type: io.kestra.plugin.core.debug.Return
    format: "v1: {{ trigger.executionId }}"

triggers:
  - id: listenFlow
    type: io.kestra.plugin.core.trigger.Flow
    inputs:
      fromParent: '{{ outputs.myTask.uri }}'
    conditions:
      - type: io.kestra.plugin.core.condition.ExecutionFlowCondition
        namespace: company.team
        flowId: trigger_flow
      - type: io.kestra.plugin.core.condition.ExecutionStatusCondition
        in:
          - SUCCESS
```

Parent flow:

```
id: trigger_flow
namespace: company.team

tasks:
  - id: myTask
    type: io.kestra.plugin.core.http.Download
    uri: https://dummyjson.com/products
```

This flow will be triggered after the successful execution of both flows `flow-a` and `flow-b` during the current day. When the conditions are met, the counter is reset and can be re-triggered during the same day. See MultipleCondition for more details

```
id: trigger-multiplecondition-listener
namespace: company.team

tasks:
  - id: onlyListener
    type: io.kestra.plugin.core.debug.Return
    format: "let's go "

triggers:
  - id: multipleListenFlow
    type: io.kestra.plugin.core.trigger.Flow
    conditions:
      - id: multiple
        type: io.kestra.plugin.core.condition.MultipleCondition
        window: P1D
        windowAdvance: P0D
```

```
    conditions:
      flow-a:
        type: io.kestra.plugin.core.condition.ExecutionFlowCondition
        namespace: company.team
        flowId: trigger-multiplecondition-flow-a
      flow-b:
        type: io.kestra.plugin.core.condition.ExecutionFlowCondition
        namespace: company.team
        flowId: trigger-multiplecondition-flow-b
```

Simply execute the two flows below to trigger `trigger-multiplecondition-listener`:

```
id: trigger-multiplecondition-flow-a
namespace: company.team

tasks:
  - id: hello
    type: io.kestra.plugin.core.log.Log
    message: Trigger A
```

```
id: trigger-multiplecondition-flow-b
namespace: company.team

tasks:
  - id: hello
    type: io.kestra.plugin.core.log.Log
    message: Trigger B
```

Example: Alerting

In this example, the Flow Trigger conditions are set to execute the flow when any workflow execution has a warning or failed status. We can configure this flow to send a notification to Slack (or any other platform) with information around the failure. Using this pattern, you can manage alerts on failure all in one place.

```
id: failure_alert_slack
namespace: system

tasks:
  - id: send_alert
    type: io.kestra.plugin.notifications.slack.SlackExecution
    url: "{{ secret('SLACK_WEBHOOK') }}"
    channel: "#general"
    executionId: "{{ trigger.executionId }}"

triggers:
  - id: on_failure
```

```
type: io.kestra.plugin.core.trigger.Flow
conditions:
  - type: io.kestra.plugin.core.condition.ExecutionStatusCondition
    in:
      - FAILED
      - WARNING
```

Check out the Blueprint here.

GitLab CI

How to use GitLab CI to create a CI/CD pipeline for your Kestra flows.

GitLab provides a solution called GitLab CI that allows you to define pipelines in YAML files to automate tests, compilation, and deployments of your applications.

Here is an example of a GitLab CI pipeline. We define the following stages: * `validate`, where we validate our flows * `deploy`, where we deploy our flows.

```yaml
stages:
  - validate
  - deploy

default:
  image:
    name: kestra/kestra:latest
    entrypoint: [""]

variables:
  KESTRA_HOST: https://kestra.io/

validate:
  stage: validate # Validate our flows server-side
  script:
    - /app/kestra flow validate ./kestra/flows --server ${KESTRA_HOST} --api-token $KESTRA_A

deploy:
  stage: deploy
  script:
    - /app/kestra flow namespace update my_namespace ./kestra/flows/prod --server ${KESTRA_H
```

Inline Scripts in Docker

Writing code directly inside your task.

To get started with a Script task, paste your custom script inline in your YAML workflow definition along with any other configuration.

```yaml
id: api_json_to_mongodb
namespace: company.team
tasks:
  - id: extract
    type: io.kestra.plugin.scripts.python.Script
    containerImage: python:3.11-slim
    beforeCommands:
      - pip install requests kestra > /dev/null
    warningOnStdErr: false
    outputFiles:
      - "*.json"
    script: |
      import requests
      import json
      from kestra import Kestra

      response = requests.get("https://api.github.com")
      data = response.json()

      with open("output.json", "w") as output_file:
          json.dump(data, output_file)

      Kestra.outputs({"status": response.status_code})

  - id: load
    type: io.kestra.plugin.mongodb.Load
    connection:
      uri: mongodb://host.docker.internal:27017/
    database: local
    collection: github
    from: "{{ outputs.extract.outputFiles['output.json'] }}"
    description: "you can start MongoDB using: docker run -d mongo"
```

The example above uses a Python script added as a multiline string into the `script` property. The script fetches data from an API and stores it as a JSON file in Kestra's internal storage using the `outputFiles` property. The `Kestra.outputs` method allows to capture additional output variables, such as the API response status code.

The `image` argument of the `docker` property allows to *optionally* specify the Docker image to use for the script. If you don't specify an image, Kestra will use the default image for the language you are using. In this example, we use the `python:3.11-slim` image.

You can also *optionally* use the `beforeCommands` property to install libraries used in your inline script. Here, we use the command `pip install requests kestra` to install `pip` packages not available in the base image `python:3.11-slim`.

The `warningOnStdErr` property allows to specify whether to set the task run to a WARNING status if the script writes to the standard error stream. By default, this property is set to `true`. Keep in ming that a script that generates an error will always set the task run to a FAILED status, regardless of the value of this property.

Inputs

Inputs allow you to make your flows more dynamic and reusable.

Instead of hardcoding values in your flow, you can use inputs to make your workflows more adaptable to change.

How to retrieve inputs

Inputs can be accessed in any task using the following expression `{{ inputs.input_name }}`.

Defining inputs

Similar to `tasks`, `inputs` is a list of key-value pairs. Each input must have a `name` and a `type`. You can also set `defaults` for each input. Setting default values for an input is always recommended, especially if you want to run your flow on a schedule.

To reference an input value in your flow, use the `{{ inputs.input_name }}` syntax.

```yaml
id: inputs_demo
namespace: company.team

inputs:
  - id: user
    type: STRING
    defaults: Rick Astley

tasks:
  - id: hello
    type: io.kestra.plugin.core.log.Log
    message: Hey there, {{ inputs.user }}
```

Try running the above flow with different values for the `user` input. You can do this by clicking on the `Execute` button in the UI, and then typing the desired value in the menu.

Inputs

Figure 1: Inputs

::alert{type="info"} The plural form of `defaults` rather than `default` has two reasons. First, `default` is a reserved keyword in Java, so it couldn't be used. Second, this property allows you to set default value for a JSON object which can simultaneously be an array defining multiple default values. ::

Here are the most common input types:

Type	Description
STRING	It can be any string value. Strings are not parsed, they are passed as-is to any task that uses them.
INT	It can be any valid integer number (without decimals).
BOOLEAN	It must be either `true` or `false`.

Check the inputs documentation for a full list of supported input types.

Parametrize your flow

In our example, we will provide the URL of the API as an input. This way, we can easily change the URL when calling the flow without having to modify the flow itself.

```yaml
id: getting_started
namespace: company.team

inputs:
  - id: api_url
    type: STRING
    defaults: https://dummyjson.com/products

tasks:
  - id: api
    type: io.kestra.plugin.core.http.Request
    uri: "{{ inputs.api_url }}"
```

To learn more about inputs, check out the full inputs documentation.

::next-link Next, let's look at `outputs` ::

Installation Guide

Install Kestra in your preferred environment.

::ChildCard{pageUrl="/docs/installation/"} ::

Namespace

Namespace is a logical grouping of flows.

Namespaces are used to organize workflows and manage access to secrets, plugin defaults and variables.

You can think of a namespace as a **folder for your flows**. Similar to folders on your file system, namespaces can be used to organize flows into logical categories. Similar to filesystems, namespaces can be indefinitely nested.

If you're looking to completely isolate environments with their own resources on the same Kestra instance, you should look at Tenants that are part of the Enterprise Edition.

Hierarchical structure when using nested namespaces

Using the dot `.` symbol, you can add a hierarchical structure to your namespaces which allows you to logically separate environments, projects, teams and departments. This way, your product, engineering, marketing, finance, and data teams can all use the same Kestra instance, while keeping their flows organized and separated. Various stakeholders can have their own child namespaces that belong to a parent namespace grouping them by environment, project, or team.

Namespace name

A namespace name can be built from alphanumerical characters, optionally separated by `.`. The hierarchy depth for namespaces is unlimited. Here are some examples of namespaces: - `project_one` - `company.project_two` - `company.team.project_three`

Using namespaces to organize flows and files

When you create a flow, you can assign a namespace to it:

```yaml
id: hello_world
namespace: company.team
tasks:
```

namespace_mgmt_1

Figure 1: namespace_mgmt_1

namespace_mgmt_2

Figure 2: namespace_mgmt_2

```
- id: log_task
  type: io.kestra.plugin.core.log.Log
  message: hi from {{ flow.namespace }}
```

::alert{type="warning"} **Note:** Once you've saved your flow, you won't be able to change its namespace. You'll need to make a new flow in order to change the namespace. ::

Here, the flow is assigned to the `marketing` namespace. This assignment of a namespace to a flow already provides a benefit of improved organization and filtering:

Additionally, you can organize your code on a namespace-level using the embedded Code editor and Namespace Files, with the option to sync those files from Git:

Namespace Tab

Starting 0.18.0, Kestra has introduced the **Namespaces** tab in the Kestra UI for OSS. In this tab, you can see all the namespaces associated with the different flows in Kestra.

You can open the details about any namespace by clicking on the name or details button to the right of that namespace.

When you select the details button for any namespace, the namespace overview page opens which details the executions of flows in that namespace.

On the top of this page, you have different tabs:

1. **Overview:** This is the default landing page of the Namespace. This page contains the dashboards and summary about the executions of different flows in this namespace.
2. **Editor:** the in-built editor where you can add/edit the namespaceFiles.
3. **Flows:** shows all the flows in the namespace. It gives a brief about each of the flows including the flow ID, labels, last execution date and last

namespace_tab

Figure 3: namespace_tab

namespace_overview

Figure 4: namespace_overview

execution status, and the execution statistics. By selecting the details button on the right of the flow, you can navigate to that flow's page.

4. **Dependencies:** shows all the flows and which ones are dependent on each other (for example through Subflows or Flow Triggers).
5. **KV Store:** manage the key-values pairs associated with this namespace. More details on KV Store can be found here.

The other tabs: Edit, Variables, Plugin Defaults, Secrets and Audit Logs are only available for Kestra EE. More details about them can be found here.

Namespace Files

Manage Namespace Files and how to use them in your flows.

What are Namespace Files

Namespace Files are files tied to a given namespace. You can think of Namespace Files as equivalent of a project in your local IDE or a copy of your Git repository.

Namespace Files can hold Python files, R or Node.js scripts, SQL queries, dbt or Terraform projects, and many more.

You can synchronize your Git repository with a specific namespace to orchestrate dbt, Terraform or Ansible, or any other project that contains code and configuration files.

Once you add any file to a namespace, you can reference it inside of your flows using the `read()` function in EVERY task or trigger from the same namespace.

For instance, if you add a SQL query called `my_query.sql` to the `queries` directory in the `company.team` namespace, you can reference it in any `Query` task or any JDBC Trigger like so: `{{ read('queries/my_query.sql') }}`.

Here is an example showing how you can use the `read()` function in a ClickHouse Trigger to read a SQL query stored as a Namespace File:

```yaml
id: jdbc_trigger
namespace: company.team

tasks:
  - id: for_each_row
    type: io.kestra.plugin.core.flow.ForEach
    values: "{{ trigger.rows }}"
    tasks:
      - id: return
        type: io.kestra.plugin.core.debug.Return
        format: "{{ json(taskrun.value) }}"

triggers:
  - id: query_trigger
    type: io.kestra.plugin.jdbc.clickhouse.Trigger
```

```
    interval: "PT5M"
    url: jdbc:clickhouse://127.0.0.1:56982/
    username: "{{ secret('CLICKHOUSE_USERNAME') }}"
    password: "{{ secret('CLICKHOUSE_PASSWORD') }}"
    sql: "{{ read('queries/my_query.sql') }}" #  The read() function reads the content of t
    fetchType: FETCH
```

::alert{type="info"} Note: we didn't have to use the `namespaceFiles.enabled: true` property — that property is only required to inject the entire directory of files from the namespace into the working directory of a script (e.g. a Python task). More on that in the subsequent sections of this page. ::

Why use Namespace Files

Namespace Files offer a simple way to organize your code and configuration files. Before Namespace Files, you had to store your code and configuration files in a Git repository and then clone that repository at runtime using the `git.Clone` task. With Namespace Files, you can store your code and configuration files directly in the Kestra's internal storage backend. That storage backend can be your local directory or an S3 bucket to ensure maximum security and privacy.

Namespace Files make it easy to: - orchestrate Python, R, Node.js, SQL, and more, without having to worry about code dependencies, packaging and deployments — simply add your code in the embedded Code Editor or sync your Git repository with a given namespace - manage your code for a given project or team in one place, even if those files are stored in different Git repositories, or even different Git providers - share your code and configuration files between workflows and team members in your organization - orchestrate complex projects that require the code to be separated into multiple scripts, queries or modules.

How to add Namespace Files

Embedded Code Editor

The easiest way to get started with Namespace Files is to use the embedded Code Editor. This allows you to easily add custom scripts, queries and configuration files along with your flow YAML configuration files.

Get started by selecting a namespace from the dropdown menu. If you type a name of a namespace that doesn't exist yet, Kestra will create it for you.

Then, add a new file, e.g., a Python script. Add a folder named `scripts` and a file called `hello.py` with the following content:

```
print("Hello from the Editor!")
```

Once you added a file, you can use it in your flow:

```yaml
id: editor
namespace: company.team

tasks:
  - id: hello
    type: io.kestra.plugin.scripts.python.Commands
    namespaceFiles:
      enabled: true
    commands:
      - python scripts/hello.py
```

The `Execute` button allows you to run your flow directly from the Code Editor. Click on the `Execute` button to run your flow. You should then see the Execution being created in a new browser tab and once you navigate to the `Logs` tab, you should see a friendly message `Hello from the Editor!` in the logs.

PushNamespaceFiles and SyncNamespaceFiles Tasks

There's 2 tasks to help you automatically manage your namespace files with Git. This allows you to sync the latest changes from a Git repository.

This example will push Namespace Files you already have in Kestra to a Git repository for you:

```yaml
id: push_to_git
namespace: system

tasks:
  - id: commit_and_push
    type: io.kestra.plugin.git.PushNamespaceFiles
    username: git_username
    password: "{{ secret('GITHUB_ACCESS_TOKEN') }}"
    url: https://github.com/git_username/scripts
    branch: dev
    namespace: company.team
    files:
      - "example.py"
    gitDirectory: _files
    commitMessage: "add namespace files"
    dryRun: true
```

This example will sync Namespace Files inside of a Git repository to your Kestra instance:

```yaml
id: sync_files_from_git
namespace: system

tasks:
```

```
  - id: sync_files
    type: io.kestra.plugin.git.SyncNamespaceFiles
    username: git_username
    password: "{{ secret('GITHUB_ACCESS_TOKEN') }}"
    url: https://github.com/git_username/scripts
    branch: main
    namespace: git
    gitDirectory: _files
    dryRun: true
```

Check out the dedicated guides for more information: - PushNamespaceFiles - SyncNamespaceFiles

GitHub Actions CI/CD

You can leverage our official GitHub Action called deploy-action to synchronize your Git repository with a given namespace. This is useful if you want to orchestrate complex Python modules, dbt projects, Terraform or Ansible infrastructure, or any other project that contains code and configuration files with potentially multiple nested directories and files.

Here is a simple example showing how you can deploy all scripts from the `scripts` directory in your Git branch to the `prod` namespace:

```
name: Kestra CI/CD
on:
  push:
    branches:
      - main
jobs:
  prod:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: deploy-scripts-to-prod
        uses: kestra-io/deploy-action@master
        with:
          resource: namespace_files
          namespace: prod
          directory: ./scripts # directory in the Git repository
          to: ./scripts # remote directory in the namespace
          server: https://demo.kestra.io/
          user: your_username
          password: ${{secrets.KESTRA_PASSWORD}}
```

::alert{type="info"} When creating a service account role for the GitHub Action in the Enterprise Edition, you need to grant the `FLOWS` permission to the Role. ::

Terraform Provider

You can use the `kestra_namespace_file` resource from the official Kestra Terraform Provider to deploy all your custom script files from a specific directory to a given Kestra namespace.

Here is a simple example showing how you can synchronize an entire directory of scripts from the directory `src` with the `company.team` namespace using Terraform:

```
resource "kestra_namespace_file" "prod_scripts" {
  for_each  = fileset(path.module, "src/**")
  namespace = "company.team"
  filename   = each.value # or "/${each.value}"
  content   = file(each.value)
}
```

Deploy Namespace Files from Git via CLI

You can also use the Kestra CLI to deploy all your custom script files from a specific directory to a given Kestra namespace. Here is a simple example showing how you can synchronize an entire directory of local scripts with the `prod` namespace using the Kestra CLI:

```
./kestra namespace files update prod /Users/anna/gh/KESTRA_REPOS/scripts --server=http://loc
```

In fact, you can even use that command directly in a flow. You can attach a schedule or a webhook trigger to automatically execute that flow anytime you push/merge changes to your Git repository, or on a regular schedule.

Here is an example of a flow that synchronizes an entire directory of local scripts with the `prod` namespace:

```yaml
id: ci
namespace: company.team

variables:
  host: http://host.docker.internal:28080/

tasks:
  - id: deploy
    type: io.kestra.plugin.core.flow.WorkingDirectory
    tasks:
      - id: clone
        type: io.kestra.plugin.git.Clone
        url: https://github.com/kestra-io/scripts
        branch: main

      - id: deploy_files
```

```
    type: io.kestra.plugin.scripts.shell.Commands
    warningOnStdErr: false
    taskRunner:
      type: io.kestra.plugin.core.runner.Process
    commands:
      - /app/kestra namespace files update prod . . --server={{vars.host}}
```

Note that the two dots in the command `/app/kestra namespace files update prod . .` indicate that we want to sync an entire directory of files cloned from the Git repository to the root directory of the `prod` namespace. If you wanted to e.g. sync that repository to the `scripts` directory, you would use the following command: `/app/kestra namespace files update prod . scripts`. The syntax of that command follows the structure:

```
/app/kestra namespace files update <namespace> <local_directory> <remote_directory>
```

To reproduce that flow, start Kestra using the following command:

```
docker run --pull=always --rm -it -p 28080:8080  kestra/kestra:latest  server local
```

Then, open the Kestra UI at `http://localhost:28080` and create a new flow with the content above. Once you execute the flow, you should see the entire directory from the `scripts` repository being synchronized with the `prod` namespace.

How to use Namespace Files in your flows

There are multiple ways to use Namespace Files in your flows. You can use the `read()` function to read the content of a file as a string, point to the file path in the supported tasks or use a dedicated task to retrieve it as an output.

Usually, pointing to a file location, rather than reading the file's content, is required when you want to use a file as an input to a CLI command, e.g. in a `Commands` task such as `io.kestra.plugin.scripts.python.Commands` or `io.kestra.plugin.scripts.node.Commands`. In all other cases, the `read()` function can be used to read the content of a file as a string e.g. in `Query` or `Script` tasks.

You can also use the `io.kestra.plugin.core.flow.WorkingDirectory` task to read namespace files there and then use them in child tasks that require reading the file path in CLI commands e.g. `python scipts/hello.py`.

The `read()` function

Note how the script in the first section used the `read()` function to read the content of the `scripts/hello.py` file as a string using the expression `"{{ read('scripts/hello.py') }}"`. It'a important to remeber that this function reads **the content of the file as a string**. Therefore, you should use that function only in tasks that expect a

string as an input, e.g., `io.kestra.plugin.scripts.python.Script` or `io.kestra.plugin.scripts.node.Script`, rather than `io.kestra.plugin.scripts.python.Commands` or `io.kestra.plugin.scripts.node.Commands`.

The `read()` function allows you to read the content of a Namespace File stored in the Kestra's internal storage backend. The `read()` function takes a single argument, which is the absolute path to the file you want to read. The path must point to a file stored in the **same namespace** as the flow you are executing.

In this example, we have a namespace file called `example.txt` that contains the text `Hello, World!`. We can print the content to the logs by using `{{ read('example.txt') }}`:

```yaml
id: files
namespace: company.team

tasks:
  - id: log
    type: io.kestra.plugin.core.log.Log
    message: "{{ read('example.txt') }}"
```

`namespaceFiles.enabled` on supported tasks

With supported tasks, such as the `io.kestra.plugin.scripts` group, we can access files using their path and enabling the task to read namespace files.

Here is a simple `weather.py` script that reads a secret to talk to a Weather Data API:

```python
import requests
api_key = '{{ secret("WEATHER_DATA_API_KEY") }}'
url = f"https://api.openweathermap.org/data/2.5/weather?q=Paris&APPID={api_key}"
weather_data = requests.get(url)
print(weather_data.json())
```

And here is the flow that uses the script:

```yaml
id: weather_data
namespace: company.team

tasks:
  - id: get_weather
    type: io.kestra.plugin.scripts.python.Commands
    namespaceFiles:
      enabled: true
      include:
        - scripts/weather.py
    taskRunner:
      type: io.kestra.plugin.scripts.runner.docker.Docker
```

```
    containerImage: ghcr.io/kestra-io/pydata:latest
    commands:
      - python scripts/weather.py
```

`namespaceFiles` property The example above uses the `include` field to only allow the `scripts/weather.py` file to be accessible by the task.

We can control what namespace files are available to our flow with the `namespaceFiles` property.

`namespaceFiles` has 3 attributes: - `enabled`: when set to true enables all files in that namespace to be visible to the task - `include`: allows you to specify files you want to be accessible by the task - `exclude`: allows you to specify files you don't want to be accessible by the task

Namespace Tasks

You can use the Namespace Tasks to upload, download and delete tasks in Kestra.

In the example below, we have a namespace file called `example.ion` that we want to convert to a csv file. We can use the `DownloadFiles` task to generate an output that contains the file so we can easily pass it dynamically to the `IonToCsv` task.

```
id: files
namespace: company.team
tasks:
  - id: namespace
    type: io.kestra.plugin.core.namespace.DownloadFiles
    namespace: company.team
    files:
      - example.ion

  - id: ion_to_csv
    type: io.kestra.plugin.serdes.csv.IonToCsv
    from: "{{ outputs.namespace.files['/example.ion'] }}"
```

Read more about the tasks below: - UploadFiles - DownloadFiles - DeleteFiles

Include / Exclude Namespace Files

You can selectively include or exclude namespace files.

Say you have multiple namespace files present: file1.txt, file2.txt, file3.json, file4.yml. You can selectively include multiple files using `include` attribute under `namespaceFiles` as shown below:

```yaml
id: include_namespace_files
namespace: company.team

tasks:
  - id: include_files
    type: io.kestra.plugin.scripts.shell.Commands
    namespaceFiles:
      enabled: true
      include:
        - file1.txt
        - file3.json
    commands:
      - ls
```

The `include_files` task will list all the included files, i.e. `file1.txt` and `file3.json` as only those got included from the namespace through `include`.

The `exclude`, on the other hand, includes all the namespace files except those specified under `exclude`.

```yaml
id: exclude_namespace_files
namespace: company.team

tasks:
  - id: exclude_files
    type: io.kestra.plugin.scripts.shell.Commands
    namespaceFiles:
      enabled: true
      exclude:
        - file1.txt
        - file3.json
    commands:
      - ls
```

The `exclude_files` task from the above flow will list `file2.txt` and `file4.yml`, i.e. all the namespace files except those that were excluded using `exclude`.

Server components

Detailed breakdown of the server components behind Kestra.

Kestra consists of multiple server components that can be scaled independently.

Each server component interacts with internal components (internal storage, queues, and repositories).

Executor

The Executor oversees all executions and certain types of tasks, such as Flowable tasks or Flow Triggers. It requires minimal resources as it doesn't perform heavy computations.

Generally speaking, the Executor never touches your data. It orchestrates workflows based on the information it receives from the Scheduler and the Queue, and it defers the execution of tasks to Workers.

If you have a large number of Executions, you can scale the Executor horizontally. However, this is rarely necessary as the Executor is very lightweight — all heavy computations are performed by Workers.

Worker

Workers execute tasks (from the Executor) and polling triggers (from the Scheduler).

Workers are highly configurable and scalable, accommodating a wide range of tasks from simple API calls to complex computations. Workers are the only server components that need access to external services in order to connect to databases, APIs, or other services that your tasks interact with.

Worker Group (EE)

In the Enterprise Edition, Worker Groups allow tasks and polling triggers to be executed on specific worker sets. They can be beneficial in various scenarios, such as using compute instances with GPUs, executing tasks on a specific OS, restricting backend access, and region-specific execution. A default worker group is recommended per tenant or namespace.

To specify a worker group for a task, use the workerGroup.key property in the task definition to point the task to a specific worker group key. If no worker group is specified, the task will be executed on the default worker group.

Scheduler

The Scheduler manages all triggers except for Flow triggers handled by the Executor. It determines when to start a flow based on trigger conditions.

Indexer

The Indexer indexes content from Kafka topics (such as the flows and executions topics) to Elasticsearch repositories.

Webserver

The Webserver is the entry point for all external communications with Kestra. It handles all REST API calls made to Kestra and serves the Kestra UI.

Setup

How to setup Kestra Enterprise Edition.

The Setup Page will guide you through the initial configuration of your instance.

When you launch Kestra Enterprise Edition for the first time, you will be prompted to configure your instance. This includes setting up your first tenant, creating your first user, and starting the Kestra UI.

Step 1: Validate Configuration

The first screen shows the main configuration of your instance. It displays: - whether `multitenancy` is enabled - whether `default tenant` is enabled — if yes, you can skip the Step 2 allowing you to create your first tenant - which `database` backend is configured (e.g, Postgres or Elasticsearch) - which `queue` backend is configured (e.g, Postgres or Kafka) - which `internal storage` backend is configured (e.g, S3, GCS, Azure Blob Storage, Minio, or local storage) - which `secret` backend is configured (e.g, Vault, AWS Secrets Manager, Elasticsearch, or not set up yet)

This step allows you to confirm whether your configuration is valid. If not, you can correct the configuration, restart the instance, and start the setup from scratch.

Step 2: Create Your First Tenant

If multitenancy is enabled, you will be prompted to create your first tenant.

If you choose to create a tenant, you will be asked to input the tenant id and tenant name, for example: - tenant id: `stage` - tenant name: `Staging Environment`.

If you enabled a default tenant, you can skip this step.

setup_page1.png

Figure 1: setup_page1.png

setup_page2.png

Figure 2: setup_page2.png

setup_page3.png

Figure 3: setup_page3.png

Step 3: Create Your First User

Now that you have your instance configured, you will be prompted to create your first user. This user will have a Super-Admin role for the instance, and will be able to manage tenants, users and roles.

Step 4: Start Kestra UI

Once your tenant and user are configured, you will be prompted to start Kestra UI. This will take you to the instance on the first tenant, logged in as the first user.

setup_page4.png

Figure 4: setup_page4.png

Stats

Manage Stats in Kestra.

The **Stats** page provides a dashboard of Kestra usage statistics, including the number of namespaces, flows, tasks, triggers, executions, and the total execution duration (in minutes). In the Enterprise Edition, this page also shows the number of users, groups and roles.

The main goal of that section is to keep security in mind — you can either **consider upgrading** to the Enterprise Edition or **activate basic authentication** for a single user directly from the UI. Here is how this page looks like in the Open Source Edition:

And here is how it looks like in the Enterprise Edition:

In both cases, when you click on the **magnifying glass icon**, you can dive into the details of the specific stats.

stats_oss

Figure 1: stats_oss

stats_ee

Figure 2: stats_ee

Develop a Task

Here are the instructions to develop a new task.

Runnable Task

::collapse{title="Here is a simple Runnable Task that reverses a string"}

```
@SuperBuilder
@ToString
@EqualsAndHashCode
@Getter
@NoArgsConstructor
@Schema(
    title = "Reverse a string",
    description = "Reverse all letters from a string"
)
public class ReverseString extends Task implements RunnableTask<ReverseString.Output> {
    @Schema(
        title = "The base string you want to reverse"
    )
    @PluginProperty(dynamic = true)
    private String format;

    @Override
    public ReverseString.Output run(RunContext runContext) throws Exception {
        Logger logger = runContext.logger();

        String render = runContext.render(format);
        logger.debug(render);

        return Output.builder()
            .reverse(StringUtils.reverse(render))
            .build();
    }

    @Builder
    @Getter
```

```
    public static class Output implements io.kestra.core.models.tasks.Output {
        @Schema(
            title = "The reverse string "
        )
        private final String reverse;
    }
}
```

::

Lets look at this one more deeply:

Class annotations

```
@SuperBuilder
@ToString
@EqualsAndHashCode
@Getter
@NoArgsConstructor
```

These are required in order to make your plugin work with Kestra. These are Lombok annotations that allow Kestra and its internal serialization to work properly.

Class declaration

```
public class ReverseString extends Task implements RunnableTask<ReverseString.Output>
```

- `ReverseString` is the name of your task, and it can be used on Kestra with `type: io.kestra.plugin.templates.ReverseString` (aka: `{{package}}.{{className}}`).
- Class must extend `Task` to be usable.
- `implements RunnableTask<ReverseString.Output>`: must implement `RunnableTask` to be discovered and must declare the output of the task `ReverseString.Output`.

Properties

```
    @PluginProperty(dynamic = true)
    private String format;
```

Declare all the properties that you can pass to the current task in a flow. For example, this will be a valid yaml for this task:

```
type: io.kestra.plugin.templates.ReverseString
format: "{{ outputs.previousTask.name }}"
```

You can declare as many properties as you want. All of these will be filled by Kestra executors.

You can use any serializable by Jackson for your properties (ex: Double, boolean, ...). You can create any class as long as the class is Serializable.

Properties validation Properties can be validated using `javax.validation.constraints.*` annotations. When the user creates a flow, your task properties will be validated before insertion and prevent wrong definition to be inserted.

The default available annotations are: `- @Positive - @AssertFalse - @AssertTrue - @Max - @Min - @Negative - @NegativeOrZero - @Positive - @PositiveOrZero - @NotBlank - @NotNull - @Null - @NotEmpty - @Past - @PastOrPresent - @Future - @FutureOrPresent`

You can also create your own custom validation. You must defined the annotation as follows:

```
@Retention(RetentionPolicy.RUNTIME)
@Constraint(validatedBy = { })
public @interface CronExpression {
    String message() default "invalid cron expression ({validatedValue})";
}
```

And you must also define a factory to inject the validation method:

```
@Factory
public class ValidationFactory {
private static final CronParser CRON_PARSER = new CronParser(CronDefinitionBuilder.instanceI

    @Singleton
    ConstraintValidator<CronExpression, CharSequence> cronExpressionValidator() {
        return (value, annotationMetadata, context) -> {
            if (value == null) {
                return true;
            }

            try {
                Cron parse = CRON_PARSER.parse(value.toString());
                parse.validate();
            } catch (IllegalArgumentException e) {
                return false;
            }

            return true;
        };
    }
}
```

Run

```
@Override
public ReverseString.Output run(RunContext runContext) throws Exception {
    Logger logger = runContext.logger();

    String render = runContext.render(format);
    logger.debug(render);

    return Output.builder()
        .reverse(StringUtils.reverse(render))
        .build();
}
```

The `run` method is where the main logic of your task will do all the work needed. You can use any Java code here with any required libraries as long as you have declared them in the Gradle configuration.

Log

```
Logger logger = runContext.logger();
```

To have a logger, you need to use this instruction. This will provide a logger for the current execution and will log appropriately. Do not create your own custom logger in order to track logs on the UI.

Render variables

```
String render = runContext.render(format);
```

In order to use dynamic expressions, you need to render them i.e. transform the properties with Pebble. Do not forget to render variables if you need to pass an output from previous variables.

You also need to add the annotation `@PluginProperty(dynamic = true)` in order to explain in the documentation that you can pass some dynamic variables. Provide a `@PluginProperty` annotation even if you didn't set any of its attributes for all variables or the generated documentation will not be accurate.

Kestra storage You can read any files from Kestra storage using the method `runContext.uriToInputStream()`

```
final URI from = new URI(runContext.render(this.from));
final InputStream inputStream = runContext.uriToInputStream(from);
```

You will get an `InputStream` in order to read the file from Kestra storage (coming from inputs or task outputs).

You can also write files to Kestra's internal storage using `runContext.putTempFile(File file)`. The local file will be deleted, so you must use a temporary file.

```
File tempFile = File.createTempFile("concat_", "");
runContext.putTempFile(tempFile)
```

Do not forget to provide Outputs with the link generated by `putTempFile` in order for it to be usable by other tasks.

Outputs

```
public class ReverseString extends Task implements RunnableTask<ReverseString.Output> {
    @Override
    public ReverseString.Output run(RunContext runContext) throws Exception {
        return Output.builder()
            .reverse(StringUtils.reverse(render))
            .build();
    }

    @Builder
    @Getter
    public static class Output implements io.kestra.core.models.tasks.Output {
        @Schema(
            title = "The reversed string"
        )
        private final String reverse;
    }
}
```

Each task must return a class instance with output values that can be used in the next tasks. You must return a class that implements `io.kestra.core.models.tasks.Output`. You can add as many properties as you want, just keep in mind that outputs need to be serializable. At execution time, outputs can be accessed by downstream tasks by leveraging outputs expressions e.g. `{{ outputs.task_id.output_attribute }}`.

If your task doesn't provide any outputs (mostly never), you use `io.kestra.core.models.tasks.VoidOutput`:

```
public class NoOutput extends Task implements FlowableTask<VoidOutput> {
    @Override
    public VoidOutput run(RunContext runContext) throws Exception {
        return null;
    }
}
```

Exception

In the `run` method, you can throw any `Exception` that will be caught by Kestra and will fail the execution. We advise you to throw any Exception that can break your task as soon as possible.

Metrics

You can expose metrics to add observability to your task. Metrics will be recorded with the execution and can be accessed via the UI or as Prometheus metrics.

There are two kinds of metrics available:

- `Counter`: `Counter.of("your.counter", count, tags);` with args
  - `String name`: The name of the metric
  - `Double|Long|Integer|Float count`: the associated counter
  - `String... tags`: a list of tags associated with your metric
- `Timer`: `Timer.of("your.duration", duration, tags);`
  - `String name`: The name of the metric
  - `Duration duration`: the recorded duration
  - `String... tags`: a list of tags associated with your metric

To save metrics with the execution, you need to use `runContext.metric(metric)`.

Name Must be lowercase separated by dots.

Tags Must be pairs of tag key and value. An example of two valid tags (`zone` and `location`) is:

```
Counter.of("your.counter", count, "zone", "EU", "location", "France");
```

Documentation

Remember to document your tasks. For this, we provide a set of annotations explained in the Document each plugin section.

Flowable Task

Flowable tasks are the most complex tasks to develop, and will usually be available from the Kestra core. You will rarely need to create a flowable task by yourself.

::alert{type="warning"} When developing such tasks, you must make it fault-tolerant as an exception thrown by a flowable task can endanger the Kestra instance and lead to inconsistencies in the flow execution. ::

Keep in mind that a flowable task will be evaluated very frequently inside the Executor and must have low CPU usage; no I/O should be done by this kind of task.

In the future, complete documentation will be available here. In the meantime, you can find all the actual Flowable tasks here to have some inspiration for Sequential or Parallel tasks development.

Task Runs

A Task Run is a single run of an individual task within an Execution, where an Execution is a single run of a flow. This means an Execution can have many Task Runs.

Each Task Run has associated data such as:

- Execution ID
- State
- Start Date
- End Date

Attempts

Each task run can have one or more attempts. Most task runs will have only one attempt, but you can configure retries for a task. If retries have been configured, a task failure will generate new attempts until the retry `maxAttempt` or `maxDuration` threshold is hit.

States

Similar to Executions, Task Runs cans be in a particular state.

State	Description
`CREATED`	The Execution or Task Run is waiting to be processed. This state usually means that the Execution is in a queue and is yet to be started.
`RUNNING`	The Execution or Task Run is currently being processed.
`SUCCESS`	The Execution or Task Run has been completed successfully.
`WARNING`	The Execution or Task Run exhibited unintended behavior, but the execution continued and was flagged with a warning.
`FAILED`	The Execution or Task Run exhibited unintended behavior that caused the execution to fail.
`RETRYING`	The Execution or Task Run is currently being retried.
`RETRIED`	The Execution or Task Run exhibited unintended behavior, stopped, and created a new execution as defined by its flow-level retry policy. The policy was set to the `CREATE_NEW_EXECUTION` behavior.

State	Description
KILLING	A command was issued that asked for the Execution or Task Run to be killed. The system is in the process of killing the associated tasks.
KILLED	Execution or Task Run was killed (upon request), and no more tasks will run.

::alert{type="info"} For a detailed overview of how each Task Run transition through different states, see the States page. ::

Expression

You can access information about a taskrun using the `{{ taskrun }}` expression.

This example returns the information from `{{ taskrun }}`:

```yaml
id: taskrun
namespace: company.team

tasks:
  - id: return
    type: io.kestra.plugin.core.debug.Return
    format: "{{ taskrun }}"
```

The logs show the following:

```
{
    "id": "61TxwXQjkXfwTd4ANK6fhv",
    "startDate": "2024-11-13T14:38:38.355668Z",
    "attemptsCount": 0
}
```

Task Run Values

Some Flowable Tasks, such as ForEach and ForEachItem, group tasks together. You can use the expression `{{ taskrun.value }}` to access the value for that task run.

In the example below, `foreach` will iterate twice over the values `[1, 2]`:

```yaml
id: loop
namespace: company.team

tasks:
  - id: foreach
    type: io.kestra.plugin.core.flow.ForEach
    values: [1, 2]
```

```
  tasks:
    - id: log
      type: io.kestra.plugin.core.log.Log
      message:
      - "{{ taskrun }}"
      - "{{ taskrun.value }}"
      - "{{ taskrun.id }}"
      - "{{ taskrun.startDate }}"
      - "{{ taskrun.attemptsCount }}"
      - "{{ taskrun.parentId }}"
```

This outputs two separate log tasks, one with `1` and the other with `2`.

Parent Task Run Values

You can also use the `{{ parent.taskrun.value }}` expression to access a task run value from a parent task within nested flowable child tasks:

```
id: loop
namespace: company.team

tasks:
  - id: foreach
    type: io.kestra.plugin.core.flow.ForEach
    values: [1, 2]
    tasks:
      - id: log
        type: io.kestra.plugin.core.log.Log
        message: "{{ taskrun.value }}"
      - id: if
        type: io.kestra.plugin.core.flow.If
        condition: "{{ true }}"
        then:
          - id: log_parent
            type: io.kestra.plugin.core.log.Log
            message: "{{ parent.taskrun.value }}"
```

This will iterate through the `log` and `if` tasks twice as there are two items in `values` property. The `log_parent` task will log the parent task run value as `1` and then `2`.

Parent vs. Parents in Nested Flowable Tasks

When using nested Flowable tasks, only the direct parent task is accessible via `taskrun.value`. To access a parent task higher up the tree, you can use the `parent` and the `parents` expressions.

The following flow shows a more complex example with nested flowable parent

tasks:

```yaml
id: each_switch
namespace: company.team

tasks:
  - id: simple
    type: io.kestra.plugin.core.log.Log
    message:
      - "{{ task.id }}"
      - "{{ taskrun.startDate }}"

  - id: hierarchy_1
    type: io.kestra.plugin.core.flow.ForEach
    values: ["caseA", "caseB"]
    tasks:
      - id: hierarchy_2
        type: io.kestra.plugin.core.flow.Switch
        value: "{{ taskrun.value }}"
        cases:
          caseA:
            - id: hierarchy_2_a
              type: io.kestra.plugin.core.debug.Return
              format: "{{ task.id }}"
          caseB:
            - id: hierarchy_2_b_first
              type: io.kestra.plugin.core.debug.Return
              format: "{{ task.id }}"

            - id: hierarchy_2_b_second
              type: io.kestra.plugin.core.flow.ForEach
              values: ["case1", "case2"]
              tasks:
                - id: switch
                  type: io.kestra.plugin.core.flow.Switch
                  value: "{{ taskrun.value }}"
                  cases:
                    case1:
                      - id: switch_1
                        type: io.kestra.plugin.core.log.Log
                        message:
                          - "{{ parents[0].taskrun.value }}"
                          - "{{ parents[1].taskrun.value }}"
                    case2:
                      - id: switch_2
                        type: io.kestra.plugin.core.log.Log
```

```
                    message:
                      - "{{ parents[0].taskrun.value }}"
                      - "{{ parents[1].taskrun.value }}"
  - id: simple_again
    type: io.kestra.plugin.core.log.Log
    message:
      - "{{ task.id }}"
      - "{{ taskrun.startDate }}"
```

The `parent` variable gives direct access to the first parent, while the `parents[INDEX]` gives you access to the parent higher up the tree.

::collapse{title="Task Run JSON Object Example"}


```
{
  "id": "5cBZ1JF8kim8fbFg13bumX",
  "executionId": "6s1egIkxu3gpzzILDnyxTn",
  "namespace": "io.kestra.tests",
  "flowId": "each-sequential-nested",
  "taskId": "1-1_return",
  "parentTaskRunId": "5ABxhOwhpd2X8DtwUPKERJ",
  "value": "s1",
  "attempts": [
    {
      "metrics": [
        {
          "name": "length",
          "tags": {
            "format": "{{task.id}} > {{taskrun.value}}   {{taskrun.startDate}}"
          },
          "value": 45.0,
          "type": "counter"
        },
        {
          "name": "duration",
          "tags": {
            "format": "{{task.id}} > {{taskrun.value}}   {{taskrun.startDate}}"
          },
          "type": "timer",
          "value": "PT0.007213673S"
        }
      ],
      "state": {
        "current": "SUCCESS",
        "histories": [
          {
            "state": "CREATED",
```

```
            "date": "2025-05-04T12:02:54.121836Z"
          },
          {
            "state": "RUNNING",
            "date": "2025-05-04T12:02:54.121841Z"
          },
          {
            "state": "SUCCESS",
            "date": "2025-05-04T12:02:54.131892Z"
          }
        ],
        "duration": "PT0.010056S",
        "endDate": "2025-05-04T12:02:54.131892Z",
        "startDate": "2025-05-04T12:02:54.121836Z"
      }
    }
  ],
  "outputs": {
    "value": "1-1_return > s1   2025-05-04T12:02:53.938333Z"
  },
  "state": {
    "current": "SUCCESS",
    "histories": [
      {
        "state": "CREATED",
        "date": "2025-05-04T12:02:53.938333Z"
      },
      {
        "state": "RUNNING",
        "date": "2025-05-04T12:02:54.116336Z"
      },
      {
        "state": "SUCCESS",
        "date": "2025-05-04T12:02:54.144135Z"
      }
    ],
    "duration": "PT0.205802S",
    "endDate": "2025-05-04T12:02:54.144135Z",
    "startDate": "2025-05-04T12:02:53.938333Z"
  }
}
```


::

taskrun_view

Figure 1: taskrun_view

Task Runs Page (EE)

::alert{type="info"} This feature is only available on the Enterprise Edition ::

If you have Kestra setup using the Kafka and Elasticsearch backend, you can view Task Runs in the UI.

It's similar to the Execution View but only shows Task Runs.

Naming Conventions

Common naming conventions to keep your flows and tasks well-organized and consistent in Kestra.

Namespace Naming Convention

We recommend using the `company.team` naming convention for namespaces to maintain a well-organized and consistent structure across your workflows. This pattern helps in the following ways: 1. Centralized governance for credentials 2. Sharing configurations across namespaces 3. Simplified Git sync

Why we recommend a `company.team` namespace structure

By having **a root namespace named after your company**, you can centrally govern plugin defaults, variables and secrets and share that configuration across all other namespaces under the company root.

Adhering to this naming convention also simplifies Git operations. You can maintain a single flow that synchronizes all workflows with Git across all namespaces under the parent namespace named after your company.

The next level of namespaces should be named after your team (e.g., `company.team`). This structure allows for centralized governance and visibility at the team level before further dividing into projects, systems, or other logical hierarchies. When syncing your code with Git, that nested structure will be reflected as nested directories in your Git repository.

Example Namespace Structure

Here is an example of how you might structure your namespaces:

- `mycompany`
  - `mycompany.marketing`
    - `mycompany.marketing.projectA`
    - `mycompany.marketing.projectB`
  - `mycompany.sales`
    - `mycompany.sales.projectC`
    - `mycompany.sales.projectD`

Should you use environment-specific namespaces?

We generally recommend against using environment-specific namespaces (e.g., `dev`, `prod`, `staging`) because it can lead to several issues such as: - **Dev and prod not fully separated**: a development workflow running out of memory could impact the production instance. - **Duplication of configurations**: you may end up duplicating configurations across environments, which can lead to inconsistencies.

It's recommended to use separate Kestra instances to separate dev and prod environments.

Summary

The `company.team` namespace structure will help you to facilitate a logical, easy to maintain hierarchy, and will make it easy to sync your workflows with Git. To reliably separate dev and prod environments, use separate Kestra instances or tenants.

ID Naming Convention

We recommend using the same convention across all IDs: - Flows - Tasks - Inputs - Outputs - Triggers

Subscript notation and valid characters in IDs

Kestra doesn't *enforce* any naming convention. For example, if you want to use the URL-style naming including hyphens, Kestra supports that. However, keep in mind that IDs for flows, tasks, inputs, outputs and triggers must match the `"^[a-zA-Z0-9][a-zA-Z0-9_-]*"` regex pattern. This means that:

- you can't use any special characters except for hyphens `-` and underscores `_`
- when using hyphens, you need to follow the format `"{{ outputs.task_id[your-custom-value].attribu }}"` when referencing that ID in output expressions; the square brackets `[]` in `[your-custom-value]` is called the subscript notation and it enables using special characters such as spaces or hyphens (as in the `kebab-case` notation) in task identifiers or output attributes.

::alert{type="info"} We recommend using the `snake_case` or `camelCase` conventions over the `kebab-case`, as they allow you to avoid the subscript notation and make your flows easier to read. ::

Snake case

Snake case is a common naming convention in programming. It's popular among Python developers in the data science, AI and data engineering domain.

Here is an example of a flow using the snake case convention to name IDs for flows, inputs, outputs, tasks, and triggers:

```yaml
id: api_python_sql
namespace: prod.marketing.attribution

inputs:
  - id: api_endpoint
    type: URL
    defaults: https://dummyjson.com/products

tasks:
  - id: fetch_products
    type: io.kestra.plugin.core.http.Request
    uri: "{{ inputs.api_endpoint }}"

  - id: transform_in_python
    type: io.kestra.plugin.scripts.python.Script
    containerImage: python:slim
    beforeCommands:
      - pip install polars
    warningOnStdErr: false
    outputFiles:
      - "products.csv"
    script: |
      import polars as pl
      data = {{ outputs.fetch_products.body | jq('.products') | first }}
      df = pl.from_dicts(data)
      df.glimpse()
      df.select(["brand", "price"]).write_csv("products.csv")

  - id: sql_query
    type: io.kestra.plugin.jdbc.duckdb.Query
    inputFiles:
      in.csv: "{{ outputs.transform_in_python.outputFiles['products.csv'] }}"
    sql: |
      SELECT brand, round(avg(price), 2) as avg_price
      FROM read_csv_auto('{{ workingDir }}/in.csv', header=True)
      GROUP BY brand
      ORDER BY avg_price DESC;
    fetchType: STORE

outputs:
  - id: final_result
    value: "{{ outputs.sql_query.uri }}"
```

```yaml
triggers:
  - id: daily_at_9am
    type: io.kestra.plugin.core.trigger.Schedule
    cron: "0 9 * * *"
```

Camel case

Camel case is another common naming convention in programming. It's popular among Java and JavaScript developers. Let's look at the same flow as above, but using the camel case convention:

```yaml
id: apiPythonSql
namespace: prod.marketing.attribution

inputs:
  - id: apiEndpoint
    type: URL
    defaults: https://dummyjson.com/products

tasks:
  - id: fetchProducts
    type: io.kestra.plugin.core.http.Request
    uri: "{{ inputs.apiEndpoint }}"

  - id: transformInPython
    type: io.kestra.plugin.scripts.python.Script
    containerImage: python:slim
    beforeCommands:
      - pip install polars
    warningOnStdErr: false
    outputFiles:
      - "products.csv"
    script: |
      import polars as pl
      data = {{outputs.fetchProducts.body | jq('.products') | first}}
      df = pl.from_dicts(data)
      df.glimpse()
      df.select(["brand", "price"]).write_csv("products.csv")

  - id: sqlQuery
    type: io.kestra.plugin.jdbc.duckdb.Query
    inputFiles:
      in.csv: "{{ outputs.transformInPython.outputFiles['products.csv'] }}"
    sql: |
      SELECT brand, round(avg(price), 2) as avgPrice
      FROM read_csv_auto('{{workingDir}}/in.csv', header=True)
```

```
      GROUP BY brand
      ORDER BY avgPrice DESC;
    store: true

outputs:
  - id: finalResult
    value: "{{ outputs.sqlQuery.uri }}"

triggers:
  - id: dailyAt9am
    type: io.kestra.plugin.core.trigger.Schedule
    cron: "0 9 * * *"
```

Both conventions are valid and it's up to you to choose the one you prefer.

Deployment Architecture

Examples of deployment architectures, depending on your needs.

Kestra is a Java application that is provided as an executable. You have many deployments options: - Docker - Kubernetes - Manual deployment

At its heart, Kestra has a plugin system allowing you to choose the dependency type that fits your needs.

You can find three example deployment architectures below.

Small-sized deployment

For small-sized deployments, you can use the Kestra standalone server, an all-in-one server component that allows running all Kestra server components in a single process. This deployment architecture has no scaling capability.

In this case, a database is the only dependency. This allows running Kestra with a minimal stack to maintain. For now, we have three databases available: - PostgreSQL - MySQL - H2

Medium-sized deployment

For medium-sized deployments, where high availability is not a strict requirement, you can use a database (Postgres or MySQL) as the only dependency. This allows running Kestra with a minimal stack to maintain. For now, we have two databases available for this kind of architecture, as H2 is not a good fit when running distributed components: - PostgreSQL - MySQL

All server components will communicate through the database.

In this deployment mode, unless all components run on the same host, you must use a distributed storage implementation like Google Cloud Storage, AWS S3, or Azure Blob Storage.

Figure 1: Kestra Standalone Architecture

Kestra Architecture

Figure 2: Kestra Architecture

Kestra High Availability Architecture

Figure 3: Kestra High Availability Architecture

High-availability deployment

To support higher throughput, and full horizontal and vertical scaling of the Kestra cluster, we can replace the database with Kafka and Elasticsearch. In this case, all the server components can be scaled without any single point of failure.

Kafka and Elasticsearch are available only in the **Enterprise Edition**.

In this deployment mode, unless all components run on the same host, you must use a distributed storage implementation like Google Cloud Storage, AWS S3, or Azure Blob Storage

Kafka

Kafka is Kestra's primary dependency in high availability mode. Each of the most important server components in the deployment must have a Kafka instance up and running. Kafka allows Kestra to be a highly scalable solution.

Kafka Executor With Kafka, the Executor is a heavy Kafka Stream application. The Executor processes all events from Kafka in the right order, keeps an internal state of the execution, and merges task run results from the Worker. It also detects dead Workers and resubmits the tasks run by a dead Worker.

As the Executor is a Kafka Stream, it can be scaled as needed (within the limits of partitions count on Kafka). Still, as no heavy computations are done in the Executor, this server component only requires a few resources (unless you have a very high rate of executions).

Kafka Worker With Kafka, the Worker is a Kafka Consumer that will process any Task Run submitted to it. Workers will receive all tasks and dispatch them internally in their Thread Pool.

It can be scaled as needed (within the limits of partitions count on Kafka) and have many instances on multiple servers, each with its own Thread Pool.

With Kafka, if a Worker is dead, the Executor will detect it and resubmit their current task run to another Worker.

Elasticsearch

Elasticsearch is Kestra's User Interface database in high availability mode, allowing the display, search, and aggregation of all Kestra's data (Flows, Executions, etc.). Elasticsearch is only used by the Webserver (API and UI).

Docker Compose

Start Kestra with a PostgreSQL database backend using a Docker Compose file.

The quickest way to a production-ready lightweight Kestra installation is to leverage Docker and Docker Compose. This guide will help you get started with Kestra using Docker.

::alert{type="info"} In order to run Kestra using `docker-compose.yml` file in production in rootless mode, please look at the `Launch Kestra in Rootless Mode` section on the Podman Compose page. ::

Before you begin

Make sure you have already installed:

- Docker
- Docker Compose

Download the Docker Compose file

Download the Docker Compose file using the following command on Linux and macOS:

```
curl -o docker-compose.yml \
https://raw.githubusercontent.com/kestra-io/kestra/develop/docker-compose.yml
```

If you're on Windows, use the following command:

```
Invoke-WebRequest -Uri "https://raw.githubusercontent.com/kestra-io/kestra/develop/docker-co
```

You can also download the Docker Compose file manually and save it as `docker-compose.yml`.

Launch Kestra

Use the following command to start the Kestra server:

```
docker-compose up -d
```

Open the URL `http://localhost:8080` in your browser to launch the UI.

Adjusting the Configuration

The command above starts a *standalone* server (all architecture components in one JVM).

The configuration will be done inside the `KESTRA_CONFIGURATION` environment variable of the Kestra container. You can update the environment variable inside the Docker compose file, or pass it via the Docker command line argument.

::alert{type="info"} If you want to extend your Docker Compose file, modify container networking, or if you have any other issues using this Docker Compose file, check the Troubleshooting Guide. ::

Use a configuration file

If you want to use a configuration file instead of the `KESTRA_CONFIGURATION` environment variable to configure Kestra you can update the default `docker-compose.yml`.

First, create a configuration file containing the `KESTRA_CONFIGURATION` environment variable defined in the `docker-compose.yml` file. You can name it `application.yaml`.

Then, update `kestra` service in the `docker-compose.yml` file to mount this file into the container and make Kestra using it via the `--config` option:

```
# [...]
  kestra:
    image: kestra/kestra:latest
    pull_policy: always
    # Note that this is meant for development only. Refer to the documentation for producti
    user: "root"
    command: server standalone --worker-thread=128 --config /etc/config/application.yaml
    volumes:
      - kestra-data:/app/storage
      - /var/run/docker.sock:/var/run/docker.sock
      - /tmp/kestra-wd:/tmp/kestra-wd
      - $PWD/application.yaml:/etc/config/application.yaml
    ports:
      - "8080:8080"
      - "8081:8081"
    depends_on:
      postgres:
        condition: service_started
```

Networking in Docker Compose

The default docker-compose file doesn't configure networking for the Kestra containers. This means that you won't be able to access any services

exposed via `localhost` on your local machine (e.g., another Docker container with a mapped port). Your machine and Docker container use a different network. To use a locally exposed service from Kestra container, you can use the `host.docker.internal` hostname or `172.17.0.1`. The `host.docker.internal` address allows you to reach your host machine's services from Kestra's container.

Alternatively, you can leverage Docker network. By default, your Kestra container will be placed in a `default` network. You can add your custom services to the `docker-compose.yml` file provided by Kestra and use the services' alias (keys from `services`) to reach them. Even better would be if you create a new network e.g. network `kestra_net` and add your services to it. Then you can add this network to the `networks` section of the `kestra` service. With this, you will have access via `localhost` to all your exposed ports.

The example below shows how you can add `iceberg-rest`, `minio` and `mc` (i.e. Minio client) to your Kestra Docker Compose file.

::collapse{title="Example"}

```yaml
version: "3"

volumes:
  postgres-data:
    driver: local
  kestra-data:
    driver: local

networks:
  kestra_net:

services:
  postgres:
    image: postgres
    volumes:
      - postgres-data:/var/lib/postgresql/data
    environment:
      POSTGRES_DB: kestra
      POSTGRES_USER: kestra
      POSTGRES_PASSWORD: k3str4
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -d $${POSTGRES_DB} -U $${POSTGRES_USER}"]
      interval: 30s
      timeout: 10s
      retries: 10
    networks:
      kestra_net:
```

```yaml
iceberg-rest:
    image: tabulario/iceberg-rest
    ports:
      - 8181:8181
    environment:
      - AWS_ACCESS_KEY_ID=admin
      - AWS_SECRET_ACCESS_KEY=password
      - AWS_REGION=us-east-1
      - CATALOG_WAREHOUSE=s3://warehouse/
      - CATALOG_IO__IMPL=org.apache.iceberg.aws.s3.S3FileIO
      - CATALOG_S3_ENDPOINT=http://minio:9000
    networks:
      kestra_net:

minio:
    image: minio/minio
    container_name: minio
    environment:
      - MINIO_ROOT_USER=admin
      - MINIO_ROOT_PASSWORD=password
      - MINIO_DOMAIN=minio
    networks:
      kestra_net:
        aliases:
          - warehouse.minio
    ports:
      - 9001:9001
      - 9000:9000
    command: ["server", "/data", "--console-address", ":9001"]

mc:
    depends_on:
      - minio
    image: minio/mc
    container_name: mc
    networks:
      kestra_net:
    environment:
      - AWS_ACCESS_KEY_ID=admin
      - AWS_SECRET_ACCESS_KEY=password
      - AWS_REGION=us-east-1
    entrypoint: >
      /bin/sh -c "
      until (/usr/bin/mc config host add minio http://minio:9000 admin password) do echo '
      /usr/bin/mc rm -r --force minio/warehouse;
      /usr/bin/mc mb minio/warehouse;
```

```
      /usr/bin/mc policy set public minio/warehouse;
      tail -f /dev/null
      "

  kestra:
    image: kestra/kestra:latest
    pull_policy: always
    entrypoint: /bin/bash
    # Note that this is meant for development only. Refer to the documentation for produc
    user: "root"
    command:
      - -c
      - /app/kestra server standalone --worker-thread=128
    volumes:
      - kestra-data:/app/storage
      - /var/run/docker.sock:/var/run/docker.sock
      - /tmp/kestra-wd:/tmp/kestra-wd
    environment:
      KESTRA_CONFIGURATION: |
        datasources:
          postgres:
            url: jdbc:postgresql://postgres:5432/kestra
            driverClassName: org.postgresql.Driver
            username: kestra
            password: k3str4
        kestra:
          server:
            basicAuth:
              enabled: false
              username: admin
              password: kestra
          repository:
            type: postgres
          storage:
            type: minio
            minio:
              endpoint: http://minio
              port: 9000
              accessKey: admin
              secretKey: password
              region: us-east-1
              bucket: warehouse
          queue:
            type: postgres
          tasks:
            tmpDir:
```

```
            path: /tmp/kestra-wd/tmp
          url: http://localhost:8080/
    ports:
      - "8080:8080"
      - "8081:8081"
    depends_on:
      postgres:
        condition: service_started
    networks:
      kestra_net:
```

::

Finally, you can also use the `host` network mode for the `kestra` service. This will make your container use your host network and you will be able to reach all your exposed ports. This means you have to change the `services.kestra.environment.KESTRA_CONFIGURATION.datasources.postgres.url` to `jdbc:postgresql://localhost:5432/kestra`. This is the easiest way but it can be a security risk.

See the example below using `network_mode: host`.

::collapse{title="Example"}

```
version: "3"

volumes:
  kestra-data:
    driver: local

services:
  kestra:
    image: kestra/kestra:latest
    pull_policy: always
    entrypoint: /bin/bash
    network_mode: host
    environment:
      JAVA_OPTS: "--add-opens java.base/java.nio=ALL-UNNAMED"
      NODE_OPTIONS: "--max-old-space-size=4096"
      KESTRA_CONFIGURATION: |
        datasources:
          postgres:
            url: jdbc:postgresql://localhost:5432/kestra
            driverClassName: org.postgresql.Driver
            username: kestra
            password: k3str4
        kestra:
          server:
```

```
  basicAuth:
    enabled: false
    username: admin
    password: kestra
anonymousUsageReport:
  enabled: true
repository:
  type: postgres
storage:
  type: local
  local:
    basePath: "/app/storage"
queue:
  type: postgres
tasks:
  tmpDir:
    path: /tmp/kestra-wd/tmp
  scripts:
    docker:
      volume-enabled: true
  defaults: # just one example to show global pluginDefaults
    - type: io.kestra.plugin.airbyte.connections.Sync
      url: http://host.docker.internal:8000/
      username: airbyte
      password: password
url: http://localhost:8080/
variables:
  envVarsPrefix: "" # to avoid requiring KESTRA_ prefix on env vars
```

::

Postgres 16 Not Compatible with 17 Error

By default, the Docker Compose template uses the latest image for postgres. However, if you initated your Kestra database on an older version of Postgres to the latest image, you might encounter the following error:

```
The data directory was initialized by PostgreSQL version 16, which is not compatible with th
```

To resolve this, you need to specify a specific tag for the postgres image in your Docker Compose file. In the example below, we specify `16` as the error above was originally initialized by version 16:

```yaml
services:
  postgres:
    image: postgres:16
```

Execution

Execute your flows and view the outcome.

Execution is a single run of a flow in a specific state.

Task Run

A task run is a single run of an individual task within an execution.

Each task run has associated data such as:

- Execution ID
- State
- Start Date
- End Date

Read more about Task Runs on the dedicated docs page.

Attempts

Each task run can have one or more attempts. Most task runs will have only one attempt, but you can configure retries for a task. If retries have been configured, a task failure will generate new attempts until the retry `maxAttempt` or `maxDuration` threshold is hit.

Outputs

Each task can generate output data that other tasks of the current flow execution can use.

These outputs can be variables or files that will be stored inside Kestra's internal storage.

Outputs are described on each task documentation page and can be seen in the **Outputs** tab of the **Execution** page.

You can read more about Outputs on the Outputs page.

bigquery_metrics

Figure 1: bigquery_metrics

Metrics

Each task can expose metrics that may be useful in understanding the internals of a task. Metrics may include, file size, number of returned rows, or query duration. You can view the available metrics for a task type on its documentation page.

Metrics can be seen in the **Metrics** tab of the **Executions** page.

Below is an example of a flow generating metrics:

```yaml
id: load_data_to_bigquery
namespace: company.team

tasks:
  - id: http_download
    type: io.kestra.plugin.core.http.Download
    uri: https://huggingface.co/datasets/kestra/datasets/raw/main/csv/orders.csv

  - id: load_biqquery
    type: io.kestra.plugin.gcp.bigquery.Load
    description: Load data into BigQuery
    autodetect: true
    csvOptions:
      fieldDelimiter: ","
    destinationTable: kestra-dev.demo.orders
    format: CSV
    from: "{{ outputs.http_download.uri }}"
```

We can see the list of metrics that the BigQuery Load task type will generate in the documentation here.

After executing the flow, you can see the metrics generated by the BigQuery Load task in the Metrics tab.

State

An Execution or a Task Run can be in a particular state.

There are multiple possible states:

State	Description
CREATED	Execution or Task Run is waiting to be processed. This state usually means that the Execution is in a queue and is yet to be started.

State	Description
`RUNNING`	The Execution or Task Run is currently being processed.
`PAUSED`	The Execution or Task Run has been paused. This status is used for two reasons: Manual validation and Delay (for a specified duration before continuing the execution).
`SUCCESS`	The Execution or Task Run has been completed successfully.
`WARNING`	The Execution or Task Run exhibited unintended behavior, but the execution continued and was flagged with a warning.
`FAILED`	The Execution or Task Run exhibited unintended behavior that caused the execution to fail.
`KILLING`	A command was issued that asked for the Execution or Task Run to be killed. The system is in the process of killing the associated tasks.
`KILLED`	The Execution or Task Run was killed (upon request), and no more tasks will run.
`RESTARTED`	This status is transitive. It is the same as `CREATED`, but for a flow that has already been executed, failed, and has been restarted.
`CANCELLED`	The Execution or Task Run has been aborted because it has reached its defined concurrency limit. The limit was set to the `CANCEL` behavior.
`QUEUED`	The Execution or Task Run has been put on hold because it has reached its defined concurrency limit. The limit was set to the `QUEUE` behavior.
`RETRYING`	The Execution or Task Run is currently being retried.
`RETRIED`	The Execution or Task Run exhibited unintended behavior, stopped, and created a new execution as defined by its flow-level retry policy. The policy was set to the `CREATE_NEW_EXECUTION` behavior.

::alert{type="info"} For a detailed overview of how each Execution and Task Run transition through different states, see the States page. ::

Execution expressions

There are a number of execution expressions which you can use inside of your flow.

Parameter	Description
`{{ execution.id }}`	The execution ID, a generated unique id for each execution.
`{{ execution.startDate }}`	The start date of the current execution, can be formatted with `{{ execution.startDate | date("yyyy-MM-dd HH:mm:ss.SSSSSS") }}`.
`{{ execution.originalId }}`	The original execution ID, this id will never change even in case of replay and keep the first execution ID.

execute_button

Figure 2: execute_button

Execute a flow from the UI

You can trigger a flow manually from the Kestra UI by clicking the `Execute` button on the flow's page. This is useful when you want to test a flow or run it on demand.

Use automatic triggers

You can add a **Schedule trigger** to automatically launch a flow execution at a regular time interval.

Alternatively, you can add a **Flow trigger** to automatically launch a flow execution when another flow execution is completed. This pattern is particularly helpful when you want to:

- Implement a centralized namespace-level error handling strategy, e.g. to send a notification when any flow execution fails in a production namespace. Check the Alerting & Monitoring section for more details.
- Decouple your flows by following an event-driven pattern, in a backwards direction (*backwards because the flow is triggered by the completion of another flow; this is in contrast to the subflow pattern, where a parent flow starts the execution of child flows and waits for the completion of each of them*).

Lastly, you can use the **Webhook trigger** to automatically launch a flow execution when a given HTTP request is received. You can leverage the `{{ trigger.body }}` variable to access the request body and the `{{ trigger.headers }}` variable to access the request headers in your flow.

To launch a flow and send data to the flow's execution context from an external system using a webhook, you can send a POST request to the Kestra API using the following URL:

`http://<kestra-host>:<kestra-port>/api/v1/executions/webhook/<namespace>/<flow-id>/<webhook-`

Here is an example:

```
http://localhost:8080/api/v1/executions/webhook/dev/hello-world/secretWebhookKey42
```

You can also pass inputs to the flow using the `inputs` query parameter.

Execute a flow via an API call

You can trigger a flow execution by calling the API directly. This is useful when you want to start a flow execution from another application or service.

Let's use the following flow as example:

```yaml
id: hello_world
namespace: company.team

inputs:
  - id: greeting
    type: STRING
    defaults: hey

tasks:
  - id: hello
    type: io.kestra.plugin.core.log.Log
    message: "{{ inputs.greeting }}"

triggers:
  - id: webhook
    type: io.kestra.plugin.core.trigger.Webhook
    key: test1234
```

Assuming that you run Kestra locally, you can trigger a flow execution by calling the `/api/v1/executions/{namespace}/{flowId}` endpoint. This example uses `curl` but you could use something else like Postman to test this too:

```
curl -X POST \
http://localhost:8080/api/v1/executions/company.team/hello_world
```

The above command will trigger an execution of the latest revision of the `hello_world` flow from the `company.team` namespace.

Execute a specific revision of a flow

If you want to trigger an execution for a specific revision, you can use the `revision` query parameter:

```
curl -X POST \
http://localhost:8080/api/v1/executions/company.team/hello_world?revision=2
```

Execute a flow with inputs

You can also trigger a flow execution with inputs by adding the `inputs` as form data (the `-F` flag in the `curl` command):

```
curl -X POST \
http://localhost:8080/api/v1/executions/company.team/hello_world \
```

```
-F greeting="hey there"
```

You can pass inputs of different types, such as `STRING`, `INT`, `FLOAT`, `DATETIME`, `FILE`, `BOOLEAN`, and more.

```
curl -v "http://localhost:8080/api/v1/executions/company.team/kestra-inputs" \
    -H "Transfer-Encoding:chunked" \
    -H "Content-Type:multipart/form-data" \
    -F string="a string"  \
    -F optional="an optional string"  \
    -F int=1  \
    -F float=1.255  \
    -F boolean=true  \
    -F instant="2023-12-24T23:00:00.000Z" \
    -F "files=@/tmp/128M.txt;filename=file"
```

Execute a flow with FILE-type inputs

You can also pass files as an input. All files must be sent as multipart form data named `files` with a header `filename=your_kestra_input_name` indicating the name of the input.

Let's look at an example to make this clearer. Suppose you have a flow that takes a JSON file as input and reads the file's content:

```
id: large_json_payload
namespace: company.team

inputs:
  - id: myCustomFileInput
    type: FILE

tasks:
  - id: hello
    type: io.kestra.plugin.scripts.shell.Commands
    inputFiles:
      myfile.json: "{{ inputs.myCustomFileInput }}"
    taskRunner:
      type: io.kestra.plugin.core.runner.Process
    commands:
      - cat myfile.json
```

Assuming you have a file `myfile.json` in the current working directory, you can invoke the flow using the following `curl` command:

```
curl -X POST -F "files=@./myfile.json;filename=myCustomFileInput" 'http://localhost:8080/api
```

::alert{type="info"} We recommend this pattern if you need to pass large payloads to a flow. Passing a large payload directly in the request body (e.g. as

`JSON`-type input or as a raw JSON webhook body) is not recommended for privacy, performance and maintenability reasons. Such large payloads would be stored directly in your Kestra's database backend, cluttering valuable storage space and leading to potential performance or privacy issues. However, if you pass it as a JSON file using a `FILE`-type input, it will be stored in internal storage (such as S3, GCS, Azure Blob), making it more performant and cost-effective to store and retrieve. ::

Execute a flow via an API call in Python

You can also use the `requests` library in Python to make requests to the Kestra API. Here's an example:

```python
import io
import requests
from requests_toolbelt.multipart.encoder import MultipartEncoder

with open("/tmp/128M.txt", 'rb') as fh:
  url = "http://kestra:8080/api/v1/executions/company.team/hello_world"
  mp_encoder = MultipartEncoder(fields={
    "string": "a string",
    "int": 1,
    "float": 1.255,
    "datetime": "2025-04-20T13:00:00.000Z",
    "files": ("file", fh, "text/plain")
  })
  result = requests.post(
      url,
      data=mp_encoder,
      headers={"Content-Type": mp_encoder.content_type},
  )
```

Get URL to follow the Execution progress

Starting from Kestra 0.19.0, the Executions endpoint additionally returns a URL allowing to follow the Execution progress from the UI. This is particularly helpful for externally triggered long-running executions that require users to follow the workflow progress. Here is how you can use it:

1) First, create a flow:

```yaml
id: myflow
namespace: company.team

tasks:
  - id: long_running_task
    type: io.kestra.plugin.scripts.shell.Commands
    commands:
```

```
      - sleep 90
    taskRunner:
      type: io.kestra.plugin.core.runner.Process
```

2) Execute the flow via an API call:

```
curl -X POST http://localhost:8080/api/v1/executions/company.team/myflow
```

You will see output similar to the following:


```
{
  "id": "1ZiZQWCHj7bf9XLtgvAxyi",
  "namespace": "company.team",
  "flowId": "myflow",
  "flowRevision": 1,
  "state": {
    "current": "CREATED",
    "histories": [
      {
        "state": "CREATED",
        "date": "2024-09-24T13:35:32.983335847Z"
      }
    ],
    "duration": "PT0.017447417S",
    "startDate": "2024-09-24T13:35:32.983335847Z"
  },
  "originalId": "1ZiZQWCHj7bf9XLtgvAxyi",
  "deleted": false,
  "metadata": {
    "attemptNumber": 1,
    "originalCreatedDate": "2024-09-24T13:35:32.983420055Z"
  },
  "url": "http://localhost:8080/ui/executions/company.team/myflow/1ZiZQWCHj7bf9XLtgvAxyi"
}
```


You can click directly on that last URL to follow the execution progress from the UI, or you can return that URL from your application to the user who initiated the flow.

Keep in mind that you need to configure the URL of your kestra instance within your configuration file to have a full URL rather than just the suffix `/ui/executions/company.team/myflow/uuid`. Here is how you can do it:

```
kestra:
  url: http://localhost:8080
```

postman webhook

Figure 3: postman webhook

Webhook vs. API call

When sending a POST request to the `/api/v1/executions/{namespace}/{flowId}` endpoint, you can send data to the flow's execution context using `inputs`. If you want to send arbitrary metadata to the flow's execution context based on some event happening in your application, you can leverage a Webhook trigger.

Here is how you can adjust the previous `hello_world` example to use the webhook trigger instead of an API call:

```yaml
id: hello_world
namespace: company.team

inputs:
  - id: greeting
    type: STRING
    defaults: hey

tasks:
  - id: hello
    type: io.kestra.plugin.core.log.Log
    message: "{{ trigger.body ?? inputs.greeting }}"

triggers:
  - id: webhook
    type: io.kestra.plugin.core.trigger.Webhook
    key: test1234
```

You can now send a POST request to the `/api/v1/executions/webhook/{namespace}/{flowId}/{webhookKe` endpoint to trigger an execution and pass any metadata to the flow using the request body. In this example, the webhook URL would be `http://localhost:8080/api/v1/executions/webhook/company.team/hello_world/test1234`.

You can test the webhook trigger using a tool like Postman or cURL. Paste the webhook URL in the URL field and a sample JSON payload in the request body. Make sure to set:

- the request method to POST
- the request body type to raw and a JSON format.

Finally, click the Send button to trigger the flow execution. You should get a response with the execution ID and status code 200 OK.

::alert{type="info"} **When to use a webhook trigger vs. an API call to create an Execution?** To decide whether to use a webhook trigger or an API

call to create an Execution, consider the following:

- Use the **webhook trigger** when you want **to send arbitrary metadata** to the flow's execution context based on some event happening in your application.
- Use the **webhook trigger** when you want to create new executions based on some **event** happening in an **external application**, such as a GitHub event (*e.g. a Pull Request is merged*) or a new record in a SaaS application, and you want to send the event metadata (header and body) to the flow to act on it.
- Use an **API call** to create an Execution when you **don't need to send any payload** (apart from `inputs`) to the flow's execution context. ::

Execute a flow from Python

You can also execute a flow using the kestra pip package. This is useful when you want to trigger a flow execution from a Python application without creating an API request from scratch as shown in the earlier example.

First, install the package:

```
pip install kestra
```

Then, you can trigger a flow execution by calling the `execute()` method. Here is an example for the same `hello_world` flow in the namespace `company.team` as above:

```python
from kestra import Flow
flow = Flow()
flow.execute('company.team', 'hello_world', {'greeting': 'hello from Python'})
```

Now imagine that you have a flow that takes a FILE-type input and reads the file's content:

```yaml
id: myflow
namespace: company.team

inputs:
  - id: myfile
    type: FILE

tasks:
  - id: print_data
    type: io.kestra.plugin.core.log.Log
    message: "file's content {{ read(inputs.myfile) }}"
```

Assuming you have a file called `example.txt` in the same directory as your Python script, you can pass a file as an input to the flow using the following

Python code:

```python
import os
from kestra import Flow

os.environ["KESTRA_HOSTNAME"] = "http://host.docker.internal:8080" # # Set this when execut

flow = Flow()
with open('example.txt', 'rb') as fh:
    flow.execute('company.team', 'myflow', {'files': ('myfile', fh, 'text/plain')})
```

Keep in mind that `files` is a tuple with the following structure: `('input_id', file_object, 'content_type')`.

Execute with ForEachItem

The `ForEachItem` task allows you to iterate over a list of items and run a subflow for each item, or for each batch containing multiple items. This is useful when you want to process a large list of items in parallel, e.g. to process millions of records from a database table or an API payload.

The `ForEachItem` task is a **Flowable** task, which means that it can be used to define the flow logic and control the execution of the flow.

Syntax:

```yaml
id: each_example
namespace: company.team
tasks:
  - id: each
    type: io.kestra.plugin.core.flow.ForEachItem
    items: "{{ inputs.file }}" # could be also an output variable {{ outputs.extract.uri }}
    inputs:
      file: "{{ taskrun.items }}" # items of the batch
    batch:
      rows: 4
      bytes: "1024"
      partitions: 2
    namespace: company.team
    flowId: subflow
    revision: 1 # optional (default: latest)
    wait: true # wait for the subflow execution
    transmitFailed: true # fail the task run if the subflow execution fails
    labels: # optional labels to pass to the subflow to be executed
      key: value
```

::collapse{title="Full Example"}

Subflow:

```yaml
id: subflow
namespace: company.team

inputs:
  - id: items
    type: STRING

tasks:
  - id: for_each_item
    type: io.kestra.plugin.scripts.shell.Commands
    taskRunner:
      type: io.kestra.plugin.core.runner.Process
    commands:
      - cat "{{ inputs.items }}"

  - id: read
    type: io.kestra.plugin.core.log.Log
    message: "{{ read(inputs.items) }}"
```

Flow that uses the `ForEachItem` task to iterate over a list of items and run the `subflow` for a batch of 10 items at a time:

```yaml
id: each_parent
namespace: company.team

tasks:
  - id: extract
    type: io.kestra.plugin.jdbc.duckdb.Query
    sql: |
      INSTALL httpfs;
      LOAD httpfs;
      SELECT *
      FROM read_csv_auto('https://huggingface.co/datasets/kestra/datasets/raw/main/csv/order
    store: true

  - id: each
    type: io.kestra.plugin.core.flow.ForEachItem
    items: "{{ outputs.extract.uri }}"
    batch:
      rows: 10
    namespace: company.team
    flowId: subflow
    wait: true
    transmitFailed: true
    inputs:
      items: "{{ taskrun.items }}"
```

::

Kubernetes

Install Kestra in a Kubernetes cluster using a Helm chart.

Helm Chart repository

We recommend Kubernetes deployment for **production** workloads, as it allows you to scale specific Kestra services as needed.

We provide an official Helm Chart to make the deployment easier.

- The chart repository is available under helm.kestra.io.
- The source code of the charts can be found in the kestra-io/helm-charts repository on GitHub.

::alert{type="info"} All image tags provided by default can be found in the Docker installation guide. ::

Install the chart

```
helm repo add kestra https://helm.kestra.io/
helm install kestra kestra/kestra
```

By default, the chart will only deploy one standalone Kestra service with one replica. This means that all Kestra server components will be deployed within a single pod. You can change that default behavior and deploy each service independently using the following Helm chart values:

```yaml
deployments:
  webserver:
    enabled: true
  executor:
    enabled: true
  indexer:
    enabled: true
  scheduler:
    enabled: true
  worker:
    enabled: true
```

```
  standalone:
    enabled: false
```

The chart can also deploy the following related services: - A Kafka cluster and Zookeeper using `kafka.enabled: true` - An Elasticsearch cluster using `elasticsearch.enabled: true` - A MinIO standalone using `minio.enabled: true` - A PostgreSQL using `postgresql.enabled: true`

The MinIO (as the internal storage backend) and PostgreSQL (as the database backend) services are enabled by default to provide a fully working setup out of the box.

::alert{type="warning"} All external services (Kafka, Elasticsearch, Zookeeper, MinIO, PostgreSQL) are deployed using unsecured configurations (no authentication, no TLS, etc.). When installing for a production environment, make sure to adjust their configurations to secure your deployment. ::

Configuration

Here is how you can adjust the Kestra configuration: - Using a Kubernetes `ConfigMap` via the `configuration` Helm value. - Using a Kubernetes `Secret` via the `secrets` Helm value.

Both must be valid YAML that will be merged as the Kestra configuration file.

Here is an example showing how to enable Kafka as the queue implementation and configure its `bootstrap.servers` property using a secret:

```
configuration:
  kestra:
    queue:
      type: kafka

secrets:
  kestra:
    kafka:
      client:
        properties:
          bootstrap.servers: "localhost:9092"
```

Docker in Docker (DinD) Worker side car

By default, Docker in Docker (DinD) is installed on the worker in the `rootless` version. This can be restricted on some environment due to security limitations.

Some solutions you may try: - On Google Kubernetes Engine (GKE), use a node pool based on `UBUNTU_CONTAINERD` that works well with Docker DinD, even rootless - Some Kubernetes clusters support only a root version of DinD;

to make your Kestra deployment work, disable the rootless version using the following Helm chart values:

```yaml
dind:
  image:
    image: docker
    tag: dind
  args:
    - --log-level=fatal
```

Troubleshooting

Docker in Docker (DinD)

If you face some issues using Docker in Docker e.g. with Script tasks using `DOCKER` runner, start troubleshooting by attaching the terminal: `docker run -it --privileged docker:dind sh`. Then, use `docker logs container_ID` to get the container logs. Also, try `docker inspect container_ID` to get more information about your Docker container. The output from this command displays details about the container, its environments, network settings, etc. This information can help you identify what might be wrong.

Docker in Docker using Helm charts

On some Kubernetes deployments, using DinD with our default Helm charts can lead to:

```
Device "ip_tables" does not exist.
ip_tables              24576  4 iptable_raw,iptable_mangle,iptable_nat,iptable_filter
modprobe: can't change directory to '/lib/modules': No such file or directory
error: attempting to run rootless dockerd but need 'kernel.unprivileged_userns_clone' (/proc
```

To fix this, use `root` to launch the DinD container by setting the following values:

```yaml
dind:
  image:
    tag: dind
  args:
    - --log-level=fatal
  securityContext:
    runAsUser: 0
    runAsGroup: 0

securityContext:
  runAsUser: 0
  runAsGroup: 0
```

Disable Docker in Docker and use Kubernetes task runner

To avoid using `root` to spin up containers via DinD, disable DinD by setting the following Helm chart values:

```yaml
dind:
  enabled: false
```

Then, set Kubernetes task runner as the default way to run script tasks:

```yaml
pluginDefaults:
  - type: io.kestra.plugin.scripts
    forced: true
    values:
      taskRunner:
        type: io.kestra.plugin.ee.kubernetes.runner.Kubernetes
        # ... your Kubernetes runner configuration
```

Kubernetes Task Runner

Run tasks as Kubernetes pods.

How to use the Kubernetes task runner

The Kubernetes task runner executes tasks in a specified Kubernetes cluster. It is useful to declare resource limits and resource requests.

Here is an example of a workflow with a task running Shell commands in a Kubernetes pod:

```yaml
id: kubernetes_task_runner
namespace: company.team

description: |
  To get the kubeconfig file, run: `kubectl config view --minify --flatten`.
  Then, copy the values to the configuration below.
  Here is how Kubernetes task runner properties (on the left) map to the kubeconfig file's p
  - clientKeyData: client-key-data
  - clientCertData: client-certificate-data
  - caCertData: certificate-authority-data
  - masterUrl: server e.g. https://docker-for-desktop:6443
  - username: user e.g. docker-desktop

inputs:
  - id: file
    type: FILE

tasks:
  - id: shell
    type: io.kestra.plugin.scripts.shell.Commands
    inputFiles:
      data.txt: "{{ inputs.file }}"
    outputFiles:
      - "*.txt"
    containerImage: centos
    taskRunner:
      type: io.kestra.plugin.ee.kubernetes.runner.Kubernetes
```

```
      config:
        clientKeyData: client-key-data
        clientCertData: client-certificate-data
        caCertData: certificate-authority-data
        masterUrl: server e.g. https://docker-for-desktop:6443
        username: user e.g. docker-desktop
    commands:
      - echo "Hello from a Kubernetes task runner!"
      - cp data.txt out.txt
```

::alert{type='info'} To deploy Kubernetes with Docker Desktop, check out this guide.

To install `kubectl`, check out this guide. ::

```
<iframe src="https://www.youtube.com/embed/9vzwCL54rVk?si=DNtDF2LaAcXSXNTu" title="YouTube v
```

File handling

If your script task has `inputFiles` or `namespaceFiles` configured, an **init container** will be added to upload files into the main container.

Similarly, if your script task has `outputFiles` configured, a **sidecar container** will be added to download files from the main container.

All containers will use an in-memory `emptyDir` volume for file exchange.

Failure scenarios

If a task is resubmitted (e.g. due to a retry or a Worker crash), the new Worker will reattach to the already running (or an already finished) pod instead of starting a new one.

Specifying resource requests for Python scripts

Some Python scripts may require more resources than others. You can specify the resources required by the Python script in the `resources` property of the task runner.

```yaml
id: kubernetes_resources
namespace: company.team

tasks:
  - id: python_script
    type: io.kestra.plugin.scripts.python.Script
    containerImage: ghcr.io/kestra-io/pydata:latest
    taskRunner:
      type: io.kestra.plugin.ee.kubernetes.runner.Kubernetes
      namespace: default
```

```
  pullPolicy: Always
  config:
    username: docker-desktop
    masterUrl: https://docker-for-desktop:6443
    caCertData: xxx
    clientCertData: xxx
    clientKeyData: xxx
  resources:
    request: # The resources the container is guaranteed to get
      cpu: "500m" # Request 1/2 of a CPU (500 milliCPU)
      memory: "128Mi" # Request 128 MB of memory
outputFiles:
  - "*.json"
script: |
  import platform
  import socket
  import sys
  import json
  from kestra import Kestra

  print("Hello from a Kubernetes runner!")

  host = platform.node()
  py_version = platform.python_version()
  platform = platform.platform()
  os_arch = f"{sys.platform}/{platform.machine()}"


  def print_environment_info():
      print(f"Host's network name: {host}")
      print(f"Python version: {py_version}")
      print(f"Platform info: {platform}")
      print(f"OS/Arch: {os_arch}")

      env_info = {
          "host": host,
          "platform": platform,
          "os_arch": os_arch,
          "python_version": py_version,
      }
      Kestra.outputs(env_info)

      filename = "environment_info.json"
      with open(filename, "w") as json_file:
          json.dump(env_info, json_file, indent=4)
```

```
    if __name__ == '__main__':
        print_environment_info()
```

::alert{type="info"} For a full list of properties available in the Kubernetes task runner, check the Kubernetes plugin documentation or explore the same in the built-in Code Editor in the Kestra UI. ::

Using plugin defaults to avoid repetition

You can use `pluginDefaults` to avoid repeating the same configuration across multiple tasks. For example, you can set the `pullPolicy` to `Always` for all tasks in a namespace:

```yaml
id: k8s_taskrunner
namespace: company.team

tasks:
  - id: parallel
    type: io.kestra.plugin.core.flow.Parallel
    tasks:
      - id: run_command
        type: io.kestra.plugin.scripts.python.Commands
        containerImage: ghcr.io/kestra-io/kestrapy:latest
        commands:
          - pip show kestra

      - id: run_python
        type: io.kestra.plugin.scripts.python.Script
        containerImage: ghcr.io/kestra-io/pydata:latest
        script: |
          import socket

          ip_address = socket.gethostbyname(hostname)
          print("Hello from AWS EKS and kestra!")
          print(f"Host IP Address: {ip_address}")

pluginDefaults:
  - type: io.kestra.plugin.scripts.python
    forced: true
    values:
      taskRunner:
        type: io.kestra.plugin.ee.kubernetes.runner.Kubernetes
        namespace: default
        pullPolicy: Always
        config:
          username: docker-desktop
```

```
masterUrl: https://docker-for-desktop:6443
caCertData: |-
  placeholder
clientCertData: |-
  placeholder
clientKeyData: |-
  placeholder
```

Guides

Below are a number of guides to help you set up the Kubernetes Task Runner on different platforms.

Google Kubernetes Engine (GKE)

Before you begin Before you start, you need to have the following: 1. A Google Cloud account. 2. A Kestra instance in a version 0.16.0 or later with Google credentials stored as secrets or environment variables within the Kestra instance.

Set up Google Cloud Inside Google Cloud, you'll need to do the following:

1. Create and select a project
2. Create a GKE Custer
3. Enable Kubernetes Engine API
4. Setup `gcloud` CLI with `kubectl`
5. Create a Service Account

::alert{type="info"} Note: To authenticate with Google Cloud, you'll need to make a Service Account and add a JSON Key to Kestra. Read more about how to do that here For GKE, we'll need to make sure we have the `Kubernetes Engine default node service account` role assigned to our Service Account. ::

Creating our Flow Here's an example flow for using the Kubernetes Task Runner with GKE. To authenticate, you need to use OAuth with a service account.

```
id: gke_task_runner
namespace: company.team

tasks:
  - id: auth
    type: io.kestra.plugin.gcp.auth.OauthAccessToken
    projectId: "projectid" # update
    serviceAccount: "{{ secret('GOOGLE_SA') }}" # update
```

```yaml
  - id: shell
    type: io.kestra.plugin.scripts.shell.Commands
    containerImage: centos
    taskRunner:
      type: io.kestra.plugin.ee.kubernetes.runner.Kubernetes
      config:
        caCertData: "{{ secret('certificate-authority-data') }}" # update
        masterUrl: https://cluster-external-endpoint # update
        username: gke_projectid_region_clustername # update
        oauthToken: "{{ outputs.auth.accessToken['tokenValue'] }}"
    commands:
      - echo "Hello from a Kubernetes task runner!"
```

You'll need to use `gcloud` CLI tool to get the credentials such as `username`, `masterUrl` and `caCertData`. You can do this by running the following command:

```
gcloud container clusters get-credentials clustername --region myregion --project projectid
```

You'll need to update the following arguments above with your own values: - `clustername` is the name of your cluster - `myregion` is the region your cluster is in, e.g. `europe-west2` - `projectid` is the id of your Google Cloud project.

Once you've run this command, you can now access your config with `kubectl config view --minify --flatten` so you can replace `caCertData`, `masterUrl` and `username`.

Amazon Elastic Kubernetes Service (EKS)

Here's an example flow for using Kubernetes Task Runner with AWS EKS. To authenticate, you need an OAuth token.

```yaml
id: eks_task_runner
namespace: company.team

tasks:
  - id: shell
    type: io.kestra.plugin.scripts.shell.Commands
    containerImage: centos
    taskRunner:
      type: io.kestra.plugin.ee.kubernetes.runner.Kubernetes
      config:
        caCertData: "{{ secret('certificate-authority-data') }}"
        masterUrl: https://xxx.xxx.region.eks.amazonaws.com
        username: arn:aws:eks:region:xxx:cluster/cluster_name
        oauthToken: xxx
    commands:
      - echo "Hello from a Kubernetes task runner!"
```

Logs

Manage Logs generated by tasks.

On the **Logs** page, you will have access to all task logs.

On here, you can filter by: - Namespace - Log level - Time period

You can search for key words too.

Kestra User Interface Logs Page

Figure 1: Kestra User Interface Logs Page

Logs Filter

Figure 2: Logs Filter

Alerting & Monitoring

Here are some best practices for alerting and monitoring your Kestra instance.

Alerting

Failure alerts are non-negotiable. When a production workflow fails, you should get notified about it as soon as possible. To implement failure alerting, you can leverage Kestra's built in notification tasks, including: - Slack - Microsoft Teams - Email

Technically, you can add custom failure alerts to each flow separately using the `errors` tasks:

```yaml
id: onFailureAlert
namespace: company.team

tasks:
  - id: fail
    type: io.kestra.plugin.core.execution.Fail

errors:
  - id: slack
    type: io.kestra.plugin.notifications.slack.SlackIncomingWebhook
    url: "{{ secret('SLACK_WEBHOOK') }}"
    payload: |
      {
        "channel": "#alerts",
        "text": "Failure alert for flow {{ flow.namespace }}.{{ flow.id }} with ID {{ execut
      }
```

However, this can lead to some boilerplate code if you start copy-pasting this `errors` configuration to multiple flows.

To implement a centralized namespace-level alerting, we instead recommend a dedicated monitoring workflow with a notification task and a Flow trigger. Below is an example workflow that automatically sends a Slack alert as soon as any flow in a namespace `company.analytics` fails or finishes with warnings.

```yaml
id: failureAlertToSlack
```

alert notification

Figure 1: alert notification

```yaml
namespace: company.monitoring

tasks:
  - id: send
    type: io.kestra.plugin.notifications.slack.SlackExecution
    url: "{{ secret('SLACK_WEBHOOK') }}"
    channel: "#general"
    executionId: "{{trigger.executionId}}"

triggers:
  - id: listen
    type: io.kestra.plugin.core.trigger.Flow
    conditions:
      - type: io.kestra.plugin.core.condition.ExecutionStatusCondition
        in:
          - FAILED
          - WARNING
      - type: io.kestra.plugin.core.condition.ExecutionNamespaceCondition
        namespace: company.analytics
        prefix: true
```

Adding this single flow will ensure that you receive a Slack alert on any flow failure in the `company.analytics` namespace. Here is an example alert notification:

::alert{type="warning"} Note that if you want this alert to be sent on failure across multiple namespaces, you will need to add an `OrCondition` to the `conditions` list. See the example below:

```yaml
id: alert
namespace: company.system

tasks:
  - id: send
    type: io.kestra.plugin.notifications.slack.SlackExecution
    url: "{{ secret('SLACK_WEBHOOK') }}"
    channel: "#general"
    executionId: "{{trigger.executionId}}"

triggers:
  - id: listen
    type: io.kestra.plugin.core.trigger.Flow
    conditions:
```

```
      - type: io.kestra.plugin.core.condition.ExecutionStatusCondition
        in:
          - FAILED
          - WARNING
      - type: io.kestra.plugin.core.condition.OrCondition
        conditions:
          - type: io.kestra.plugin.core.condition.ExecutionNamespaceCondition
            namespace: company.product
            prefix: true
          - type: io.kestra.plugin.core.condition.ExecutionFlowCondition
            flowId: cleanup
            namespace: company.system
```

::

The example above is correct. However, if you instead list the conditions without the `OrCondition`, no alerts would be sent as kestra would try to match all criteria and there would be no overlap between the two conditions (they would cancel each other out). See the example below:

```
id: bad_example
namespace: company.monitoring
description: This example will not work

tasks:
  - id: send
    type: io.kestra.plugin.notifications.slack.SlackExecution
    url: "{{ secret('SLACK_WEBHOOK') }}"
    channel: "#general"
    executionId: "{{trigger.executionId}}"

triggers:
  - id: listen
    type: io.kestra.plugin.core.trigger.Flow
    conditions:
      - type: io.kestra.plugin.core.condition.ExecutionStatusCondition
        in:
          - FAILED
          - WARNING
      - type: io.kestra.plugin.core.condition.ExecutionNamespaceCondition
        namespace: company.product
        prefix: true
      - type: io.kestra.plugin.core.condition.ExecutionFlowCondition
        flowId: cleanup
        namespace: company.system
```

Here, there's no overlap between the two conditions. The first condition will only match executions in the `company.product` namespace, while the second condi-

tion will only match executions from the `cleanup` flow in the `company.system` namespace. If you want to match executions from the `cleanup` flow in the `company.system` namespace **or** any execution in the `product` namespace, make sure to add the `OrCondition`.

Monitoring

By default, Kestra exposes a monitoring endpoint on port 8081. You can change this port using the `endpoints.all.port` property in the configuration options.

This monitoring endpoint provides invaluable information for troubleshooting and monitoring, including Prometheus metrics and several Kestra's internal routes. For instance, the `/health` endpoint exposed by default on port 8081 (e.g. http://localhost:8081/health) generates a similar response as shown below as long as your Kestra instance is healthy:


```
{
  "name": "kestra",
  "status": "UP",
  "details": {
    "jdbc": {
      "name": "kestra",
      "status": "UP",
      "details": {
        "jdbc:postgresql://postgres:5432/kestra": {
          "name": "kestra",
          "status": "UP",
          "details": {
            "database": "PostgreSQL",
            "version": "15.3 (Debian 15.3-1.pgdg110+1)"
          }
        }
      }
    },
    "compositeDiscoveryClient()": {
      "name": "kestra",
      "status": "UP",
      "details": {
        "services": {

        }
      }
    },
    "service": {
      "name": "kestra",
      "status": "UP"
    },
```

```
    "diskSpace": {
      "name": "kestra",
      "status": "UP",
      "details": {
        "total": 204403494912,
        "free": 13187035136,
        "threshold": 10485760
      }
    }
  }
}
```


Prometheus

Kestra exposes Prometheus metrics on the endpoint `/prometheus`. This endpoint can be used by any compatible monitoring system.

For more details about Prometheus setup, refer to the Monitoring with Grafana & Prometheus article.

Kestra's metrics

You can leverage Kestra's internal metrics to configure custom alerts. Each metric provides multiple time series with tags allowing to track at least namespace & flow but also other tags depending on available tasks.

Kestra metrics use the prefix `kestra`. This prefix can be changed using the `kestra.metrics.prefix` property in the configuration options.

Each task type can expose custom metrics that will be also exposed on Prometheus.

Worker

Metrics	Type	Description
worker.running.count	`GAUGE`	Count of tasks actually running
worker.started.count	`COUNTER`	Count of tasks started
worker.retried.count	`COUNTER`	Count of tasks retried
worker.ended.count	`COUNTER`	Count of tasks ended
worker.ended.duration	`TIMER`	Duration of tasks ended
worker.job.running	`GAUGE`	Count of currently running worker jobs
worker.job.pending	`GAUGE`	Count of currently pending worker jobs
worker.job.thread	`GAUGE`	Total worker job thread count

::alert{type="info"} The `worker.job.pending`, `worker.job.running`, and `worker.job.thread` metrics are intended for autoscaling worker servers. ::

Executor

Metrics	Type	Description
executor.taskrun.next.count	COUNTER	Count of tasks found
executor.taskrun.ended.count	COUNTER	Count of tasks ended
executor.taskrun.ended.duration	TIMER	Duration of tasks ended
executor.workertaskresult.count	COUNTER	Count of task results sent by a worker
executor.execution.started.count	COUNTER	Count of executions started
executor.execution.end.count	COUNTER	Count of executions ended
executor.execution.duration	TIMER	Duration of executions ended

Indexer

Metrics	Type	Description
indexer.requests.count	COUNTER	Count of index requests sent to a repository
indexer.requests.duration	DURATION	Duration of index requests sent to a repository

Scheduler

Metrics	Type	Description
scheduler.trigger.count	COUNTER	Count of triggers
scheduler.evaluate.running.count	COUNTER	Count of triggers actually running
scheduler.evaluate.duration	TIMER	Duration of trigger evaluation

Others metrics

Kestra also exposes all internal metrics from the following sources:

- Micronaut
- Kafka
- Thread pools of the application
- JVM

Check out the Micronaut documentation for more information.

Grafana and Kibana

Kestra uses Elasticsearch to store all executions and metrics. Therefore, you can easily create a dashboard with Grafana or Kibana to monitor the health of your Kestra instance.

We'd love to see what dashboards you will build. Feel free to share a screenshot or a template of your dashboard with the community.

Kestra endpoints

Kestra exposes internal endpoints on the management port (8081 by default) to provide status corresponding to the server type:

- `/worker`: will expose all currently running tasks on this worker.
- `/scheduler`: will expose all currently scheduled flows on this scheduler with the next date.
- `/kafkastreams`: will expose all Kafka Streams states and aggregated store lag.
- `/kafkastreams/{clientId}/lag`: will expose details lag for a `clientId`.
- `/kafkastreams/{clientId}/metrics`: will expose details metrics for a `clientId`.

Other Micronaut default endpoints

Since Kestra is based on Micronaut, the default Micronaut endpoints are enabled by default on port 8081:

- `/info` Info Endpoint with git status information.
- `/health` Health Endpoint usable as an external heathcheck for the application.
- `/loggers` Loggers Endpoint allows changing logger level at runtime.
- `/metrics` Metrics Endpoint metrics in JSON format.
- `/env` Environment Endpoint to debug configuration files.

You can disable some endpoints following the above Micronaut configuration.

Debugging techniques

Without any order, here are debugging techniques that administrators can use to understand their issues:

Enable verbose log

Kestra had some management endpoints including one that allows changing logging verbosity at run time.

Inside the container (or in local if standalone jar is used), send this command to enable very verbose logging:

```
curl -i -X POST -H "Content-Type: application/json" \
  -d '{ "configuredLevel": "TRACE" }' \
  http://localhost:8081/loggers/io.kestra
```

Alternatively, you can change logging levels on configuration files:

```yaml
logger:
  levels:
    io.kestra.core.runners: TRACE
```

Take a thread dump

You can request a thread dump via the `/threaddump` endpoint available on the management port (8081 if not configured otherwise).

Outputs

Outputs allow you to pass data between tasks and flows.

Tasks and flows can generate outputs, which can be passed to downstream processes. These outputs can be variables or files stored in the internal storage.

How to retrieve outputs

Use the syntax `{{ outputs.task_id.output_property }}` to retrieve a specific output of a task.

If your task id contains one or more hyphens (i.e. the `-` sign), wrap the task id in square brackets, e.g. `{{ outputs['task-id'].output_property }}`.

To see which outputs have been generated during a flow execution, go to the **Outputs** tab on the Execution page: Output of our previous download

The outputs are useful for troubleshooting and auditing. Additionally, you can use outputs to: - share **downloadable artifacts** with business stakeholders, e.g. a table generated by a SQL query, or a CSV file generated by a Python script - **pass data** between decoupled processes (e.g. pass subflow's outputs or a file detected by S3 trigger to downstream tasks)

Use outputs in your flow

When fetching data from a REST API, Kestra stores that fetched data in the internal storage, and makes it available to downstream tasks using the `body` output argument.

Use the `{{ outputs.task_id.body }}` syntax to process that fetched data in a downstream task, as shown in the Python script task below.

```yaml
id: getting_started_output
namespace: company.team

inputs:
  - id: api_url
    type: STRING
```

Debug Outputs

Figure 1: Debug Outputs

```
    defaults: https://dummyjson.com/products

tasks:
  - id: api
    type: io.kestra.plugin.core.http.Request
    uri: "{{ inputs.api_url }}"

  - id: python
    type: io.kestra.plugin.scripts.python.Script
    containerImage: python:slim
    beforeCommands:
      - pip install polars
    warningOnStdErr: false
    outputFiles:
      - "products.csv"
    script: |
      import polars as pl
      data = {{outputs.api.body | jq('.products') | first}}
      df = pl.from_dicts(data)
      df.glimpse()
      df.select(["brand", "price"]).write_csv("products.csv")
```

This flow processes data using Polars and stores the result as a CSV file.

::alert{type="info"} To avoid package dependency conflicts, the Python task is running in an **independent Docker container**. You can optionally provide a **custom Docker image** from a private container registry, or use a public Python image from DockerHub, and install any custom package dependencies using the `beforeCommands` argument. The `beforeCommands` argument allows you to install any custom package dependencies — here, we install Polars. Use as many commands as needed to prepare containerized environment for the script execution. ::

Debug Outputs

When referencing the output from the previous task, this flow uses `jq` language to extract the `products` array from the API response — `jq` is available in all Kestra tasks without having to install it.

You can test `{{ outputs.task_id.body | jq('.products') | first }}` and any other output parsing expression using the built-in expressions evaluator on the **Outputs** page:

Passing data between tasks

Let's add another task to the flow to process the CSV file generated by the Python script task. We will use the `io.kestra.plugin.jdbc.duckdb.Query` task to run a SQL query on the CSV file and store the result as a downloadable artifact in the internal storage.

```yaml
id: getting_started
namespace: company.team

tasks:
  - id: api
    type: io.kestra.plugin.core.http.Request
    uri: https://dummyjson.com/products

  - id: python
    type: io.kestra.plugin.scripts.python.Script
    containerImage: python:slim
    beforeCommands:
      - pip install polars
    warningOnStdErr: false
    outputFiles:
      - "products.csv"
    script: |
      import polars as pl
      data = {{ outputs.api.body | jq('.products') | first }}
      df = pl.from_dicts(data)
      df.glimpse()
      df.select(["brand", "price"]).write_csv("products.csv")

  - id: sqlQuery
    type: io.kestra.plugin.jdbc.duckdb.Query
    inputFiles:
      in.csv: "{{ outputs.python.outputFiles['products.csv'] }}"
    sql: |
      SELECT brand, round(avg(price), 2) as avg_price
      FROM read_csv_auto('{{ workingDir }}/in.csv', header=True)
      GROUP BY brand
      ORDER BY avg_price DESC;
    store: true
```

This example flow passes data between tasks using outputs. The `inputFiles` argument of the `io.kestra.plugin.jdbc.duckdb.Query` task allows you to pass files from internal storage to the task. The `store: true` ensures that the result of the SQL query is stored in the internal storage and can be previewed

Preview

Figure 2: Preview

and downloaded from the Outputs tab.

To sum up, our flow extracts data from an API, uses that data in a Python script, executes a SQL query and generates a downloadable artifact.

::alert{type="info"} If you encounter any issues while executing the above flow, this might be a Docker related issue (e.g. Docker-in-Docker setup, which might be difficult to configure on Windows). Set the runner property to `PROCESS` to run the Python script task in the same process as the flow rather than in a Docker container, as shown in the example below. This will avoid any Docker related issues. ::

```yaml
id: getting_started
namespace: company.team

inputs:
  - id: api_url
    type: STRING
    defaults: https://dummyjson.com/products

tasks:
  - id: api
    type: io.kestra.plugin.core.http.Request
    uri: "{{ inputs.api_url }}"

  - id: python
    type: io.kestra.plugin.scripts.python.Script
    taskRunner:
      type: io.kestra.plugin.core.runner.Process
    beforeCommands:
      - pip install polars
    warningOnStdErr: false
    outputFiles:
      - "products.csv"
    script: |
      import polars as pl
      data = {{ outputs.api.body | jq('.products') | first }}
      df = pl.from_dicts(data)
      df.glimpse()
      df.select(["brand", "price"]).write_csv("products.csv")

  - id: sqlQuery
    type: io.kestra.plugin.jdbc.duckdb.Query
```

```
inputFiles:
  in.csv: "{{ outputs.python.outputFiles['products.csv'] }}"
sql: |
  SELECT brand, round(avg(price), 2) as avg_price
  FROM read_csv_auto('{{ workingDir }}/in.csv', header=True)
  GROUP BY brand
  ORDER BY avg_price DESC;
store: true
```

To learn more about outputs, check out the full outputs documentation.

::next-link Next, let's cover `triggers` to schedule the flow ::

Revision

Manage versions of flows.

```
<iframe src="https://www.youtube.com/embed/lpHl52Rlvr0?si=RyPvvhGNkTmskLKP" title="YouTube v
```

Flows are versioned by default. Whenever you make any changes to your flows, a new revision is created. This allows you to rollback to a previous version of your flow if needed.

If you navigate to a specific flow and go to the **Revisions** tab, you will see a list of all revisions of that flow. You can then compare the differences between two revisions side-by-side or line-by-line and rollback to a previous revision if needed.

Figure 1: revisions

DOCKER and PROCESS runners

Manage the environment your code is executed with `runner`.

Kestra supports two runners for scripting tasks: `DOCKER` and `PROCESS`.

::alert{type="info"} The `runner` property is being replaced with Task Runners to give you more control and flexibility. Read more about Task Runners here. ::

You can configure your scripts to run either in local **processes** or in **Docker containers** by using the `runner` property:

1. By default all scripting tasks run in isolated containers using the `DOCKER` runner.
2. Setting the `runner` property to `PROCESS` will execute your task in a local process on the worker without relying on Docker for container isolation.

runner: DOCKER

Docker is the default option for all script tasks. There are many arguments that can be provided here, including credentials to private Docker registries:

```yaml
id: python_in_container
namespace: company.team

tasks:
  - id: wdir
    type: io.kestra.plugin.core.flow.WorkingDirectory
    tasks:
      - id: cloneRepository
        type: io.kestra.plugin.git.Clone
        url: https://github.com/kestra-io/examples
        branch: main

      - id: gitPythonScripts
        type: io.kestra.plugin.scripts.python.Commands
        warningOnStdErr: false
        outputFiles:
          - "*.csv"
```

```
      - "*.parquet"
    commands:
      - python scripts/etl_script.py
    runner: DOCKER
    docker:
      image: annageller/kestra:latest
      config: |
        {
          "auths": {
              "https://index.docker.io/v1/": {
                  "username": "annageller",
                  "password": "{{ secret('DOCKER_PAT') }}"
              }
          }
        }
```

Head over to the Secrets section to learn more about secrets in Kestra.

runner: PROCESS

The `PROCESS` runner is useful if your Kestra instance is running locally without Docker and you want to access your local files and environments, for example, to take advantage of locally configured Conda virtual environments.

```
id: local_python_script
namespace: company.team

tasks:
  - id: conda_example
    type: io.kestra.plugin.scripts.python.Commands
    runner: PROCESS
    beforeCommands:
      - conda activate myCondaEnv
    commands:
      - python /Users/you/scripts/etl_script.py
```

Running scripts in a local process is particularly beneficial when using remote Worker Groups. The example below ensures that a script will be picked up only by Kestra workers that have been started with the key `gpu`, effectively delegating processing of scripts that require demanding computational requirements to the right server, rather than running them directly in a local container:

```
id: gpu_task
namespace: company.team

tasks:
  - id: gpu
    type: io.kestra.plugin.scripts.python.Commands
```

```yaml
runner: PROCESS
commands:
  - python ml_on_gpu.py
workerGroup:
  key: gpu
```

Task Runner vs. Worker Group

Find out when to use Task Runners or Worker Groups.

Overview

Task Runners and Worker Groups both **offload compute-intensive tasks to dedicated workers**. However, **worker groups have a broader scope**, applying to **all tasks** in Kestra, whereas **task runners** are limited to **scripting tasks** (Python, R, JavaScript, Shell, dbt, etc. — see the full list here). Worker groups can be used with any plugins.

For instance, if you need to query an on-premise SQL Server database running on a different server than Kestra, your SQL Server Query task can target a worker with access to that server. Additionally, worker groups can fulfill the same use case as task runners by distributing the load of scripting tasks to dedicated workers with the necessary resources and dependencies (*incl. hardware, region, network, operating system*).

Key differences

Worker groups are always-on servers that can run any task in Kestra, while task runners are ephemeral containers that are spun up only when a task is executed. This has implications with respect to latency and cost: - Worker groups are running on dedicated servers, so they can start executing tasks immediately with millisecond latency. Task runners, on the other hand, need to be spun up before they can execute a task, which can introduce latency up to minutes. For example, the AWS Batch task runner can take up to 50 seconds to register a task definition and start a container on AWS ECS Fargate. With the Google Batch task runner, it can take up to 90 seconds if you don't use a compute reservation because GCP spins up a new compute instance for each task run. - Task runners can be more cost-effective for infrequent short-lived tasks, while worker groups are more cost-effective for frequent and long-running tasks. - Worker Groups work at the task level whereas Task Runner is only available for some task types such as Scripts, Commands, CLI.

The table below summarizes the differences between task runners and worker groups.

	Task Runners	Worker Groups
Scope	Limited to scripting tasks	Applicable to all tasks in Kestra
Use Cases	Scripting tasks (Python, R, etc.)	Any task, including database queries
Deployment	Ephemeral containers	Always-on servers
Resource Handling	Spins up as needed	Constantly available
Latency	High latency (seconds, up to minutes)	Low latency (milliseconds)
Cost Efficiency	Suitable for infrequent tasks	Suitable for frequent or long-running tasks

Use cases

Here are common use cases in which **Worker Groups** can be beneficial: - Execute tasks and polling triggers on specific servers (e.g., a VM with access to your on-premise database or a server with preconfigured CUDA drivers). - Execute tasks and polling triggers on a worker with a specific Operating System (e.g., a Windows server configured with specific software needed for a task). - Restrict backend access to a set of workers (firewall rules, private networks, etc.).

Here are common use cases in which **Task Runners** can be beneficial: - Offload compute-intensive tasks to compute resources provisioned on-demand. - Run tasks that temporarily require more resources than usual e.g., during a backfill or a nightly batch job. - Run tasks that require specific dependencies or hardware (e.g., GPU, memory, etc.).

Usage

Worker Groups Usage

First, make sure you start the worker with the `--worker-group myWorkerGroupKey` flag. It's important for the new worker to have a configuration similar to that of your principal Kestra server and to have access to the same backend database and internal storage. The configuration file will be passed via the `--config` flag, as shown in the example below.

```
kestra server worker --worker-group=myWorkerGroupKey --config=/path/to/kestra-config.yaml
```

To assign a task to the desired worker group, simply add a `workerGroup.key` property. This will ensure that the task or polling trigger is executed on a worker in the specified worker group.

```yaml
id: myflow
namespace: company.team
```

default_worker_group

Figure 1: default_worker_group

```yaml
tasks:
  - id: gpu
    type: io.kestra.plugin.scripts.python.Commands
    namespaceFiles:
      enabled: true
    commands:
      - python ml_on_gpu.py
    workerGroup:
      key: myWorkerGroupKey
```

A default worker group can also be configured at the namespace level so that all tasks and polling triggers in that namespace are executed on workers in that worker group by default.

Task Runners Usage

To use a task runner, add a `taskRunner` property to your task configuration and choose the desired `type` of task runner. For example, to use the AWS Batch task runner, you would configure your task as follows:

```yaml
id: aws_ecs_fargate_python
namespace: company.team

tasks:
  - id: run_python
    type: io.kestra.plugin.scripts.python.Script
    containerImage: ghcr.io/kestra-io/pydata:latest
    taskRunner:
      type: io.kestra.plugin.ee.aws.runner.Batch
      computeEnvironmentArn: "arn:aws:batch:eu-west-1:707969873520:compute-environment/kestr
      jobQueueArn: "arn:aws:batch:eu-west-1:707969873520:job-queue/kestraJobQueue"
      executionRoleArn: "arn:aws:iam::707969873520:role/kestraEcsTaskExecutionRole"
      taskRoleArn: "arn:aws:iam::707969873520:role/ecsTaskRole"
      accessKeyId: "{{ secret('AWS_ACCESS_KEY_ID') }}"
      secretKeyId: "{{ secret('AWS_SECRET_ACCESS_KEY') }}"
      region: eu-west-1
      bucket: kestra-ie
    script: |
      import platform
      import socket
      import sys
```

```
def print_environment_info():
    print("Hello from AWS Batch and kestra!")
    print(f"Host's network name: {platform.node()}")
    print(f"Python version: {platform.python_version()}")
    print(f"Platform information (instance type): {platform.platform()}")
    print(f"OS/Arch: {sys.platform}/{platform.machine()}")

    try:
        hostname = socket.gethostname()
        ip_address = socket.gethostbyname(hostname)
        print(f"Host IP Address: {ip_address}")
    except socket.error as e:
        print("Unable to obtain IP address.")


if __name__ == '__main__':
    print_environment_info()
```

Tenants

How to enable multi-tenancy in your Kestra instance.

What is Multi-Tenancy

A tenant represents an **isolated environment within a single Kestra instance**.

Each tenant functions as a separate entity with its own resources, such as flows, triggers, or executions. Multi-tenancy enables different teams, projects, or customers to operate independently within the same Kestra instance, ensuring data privacy, security along with separation of resources between business units, teams, or customers. For example, you can have a `dev` tenant for development, a `staging` tenant for testing, and a `prod` tenant for production.

::alert{type="info"} You can think of multi-tenancy as running multiple virtual instances in a single physical instance of Kestra Cloud or Kestra Enterprise Edition. ::

When multi-tenancy is enabled, all resources (such as flows, triggers, executions, RBAC, and more) are isolated by the tenant. This means that you can have a flow with the same identifier and the same namespace in multiple tenants at the same time.

Data stored inside the internal storage are also isolated by tenants.

Multi-tenancy functionality is not visible to end-users from the UI except for the tenant selection dropdown menu. That dropdown menu lists all tenants a user has access to, allowing them to switch between tenants easily. Each UI page also includes the tenant ID in the URL e.g. `https://demo.kestra.io/ui/yourTenantId/executions/namespace/flow/executionId`.

The API URLs also include the tenant identifier. For example, the URL of the API operation to list flows of the `marketing` namespace is `/api/v1/flows/marketing` when multi-tenancy is not enabled. Once

Figure 1: Tenants selection dropdown

you enable multi-tenancy and create a tenant `prod`, this URL becomes `/api/v1/prod/flows/marketing`. You can check the API Guide for more information.

Tenants must be created upfront, and a user needs to be granted access to use a specific tenant.

Key Benefits of Multi-Tenancy

1. **Data Isolation**: each tenant's data, configuration and code are isolated and inaccessible to other tenants.
2. **Resource Isolation**: each tenant's resources are isolated from other tenants — incl. flows, triggers, executions, logs, audit logs, secrets, etc.
3. **Simple Configuration**: you can easily create new tenants instantly giving you a fresh, fully-isolated workspace accessible from your existing Kestra instance.
4. **Intuitive UI Navigation**: the UI provides a dropdown as well as tenant identifiers included in the URL to make switching between tenants seamless.

How to Enable Multi-Tenancy

By default, multi-tenancy is disabled. To enable it, add the following configuration:

```
kestra:
  ee:
    tenants:
      enabled: true
```

::alert{type="warning"} If you enable multi-tenancy in a Kestra instance with existing resources (flows, namespaces, executions), make sure to execute the `kestra auths users sync-access` command to synchronize the existing access permissions with the default tenant. ::

Default Tenant

When enabling multi-tenancy, you can also decide whether to enable the default tenant or not. Default tenant is a tenant without an identifier (aka the null tenant). It exists for backward compatibility when multi-tenancy is enabled in an existing Kestra instance. If you disable the default tenant in a Kestra instance that already has flows and executions, you will no longer be able to access them.

When multi-tenancy is enabled in a new Kestra instance, **it's recommended to disable the default tenant** so that all tenants will have an identifier. This way, all tenants are explicitly defined and can be referenced by their ID.

By default, multi-tenancy is disabled, and the default tenant is set to `true`. Once you enable multi-tenancy, you can set the **default tenant** to `false` to disable it so that your Kestra instance includes only the tenants you explicitly create. Here is how to enable multi-tenancy and disable the default tenant (best practice):

```yaml
kestra:
  ee:
    tenants:
      enabled: true
      defaultTenant: false
```

::alert{type="info"} Note that in Kestra Cloud, multi-tenancy is automatically enabled and the default tenant is disabled. ::

Once multi-tenancy is enabled, you can create one or more tenants e.g. from the UI, CLI, Terraform or API. Then, you can assign user roles and permissions within each tenant.

Creating and Managing Tenants

Tenants in Kestra can be managed in various ways: from the UI, CLI, API, or Terraform.

Creating a Tenant from the UI

Tenants can be created and managed directly through Kestra's user interface. Go to **Administration -> Tenants**. Then, click on the **Create** button: create tenant from the UI

Fill in the form and click **Save**: create tenant from the UI

The user who created the tenant will get an Admin Role for that tenant. You may need to refresh the UI to see updated Roles.

Creating a Tenant from the CLI

Kestra provides CLI commands for tenant creation. The following command will create a tenant with the identifier `stage` and the name `Staging`:

```
kestra tenants create --tenant stage --name "Staging"
```

Running `kestra tenants create --help` will show you all available properties:

```
$ kestra tenants create --help
Usage: kestra tenants create [-hVv] [--internal-log]
                             [--admin-username=<adminUser>] [-c=<config>]
                             [-l=<logLevel>] [--name=<tenantName>]
                             [-p=<pluginsPath>] [--tenant=<tenantId>]
```

```
create a tenant and assign admin roles to an existing admin user
      --admin-username=<adminUser>
                              Username of an existing admin user that will be
                                admin of this tenant
  -c, --config=<config>       Path to a configuration file, default: /Users/anna/.
                                kestra/config.yml)
  -h, --help                  Show this help message and exit.
      --internal-log          Change also log level for internal log, default:
                                false)
  -l, --log-level=<logLevel>
                              Change log level (values: TRACE, DEBUG, INFO, WARN,
                                ERROR; default: INFO)
      --name=<tenantName>     tenant description
  -p, --plugins=<pluginsPath>
                              Path to plugins directory , default:
                                /Users/anna/dev/plugins)
      --tenant=<tenantId>     tenant identifier
  -v, --verbose               Change log level. Multiple -v options increase the
                                verbosity.
  -V, --version               Print version information and exit.
```

Creating a Tenant from the API

Tenants can be managed programmatically via Kestra's API. Here is an example of an API call for creating a tenant:

```
curl -X POST "https://demo.kestra.io/api/v1/tenants" \
  -H "accept: application/json" \
  -H "Content-Type: application/json" \
  -d "{ \"id\": \"stage\", \"name\": \"staging\", \"deleted\": false}"
```

Creating a Tenant from Terraform

Tenants can be managed via Infrastructure as Code using Kestra's Terraform provider. Here is an example of a Terraform configuration for creating a tenant:

```
resource "kestra_tenant" "stage" {
  tenant_id = "stage"
  name      = "staging"
}
```

Admin Role Assignment

Regardless of which of the above methods you use to create a tenant, the User who creates the tenant will automatically get the Admin Role assigned. That role grants admin rights to that user on that tenant.

Note that there is an exception to this rule if tenant is created by a Super Admin.

In that case, the Super Admin will have to explicitly assign the Admin Role for that tenant to themselves or any other User, Service Account or Group.

Terraform

How to use Terraform to provision and manage changes to Kestra resources.

Kestra Provider

You can use the Official Kestra Provider to provision and manage changes to Kestra resources.

Multitenancy

The Kestra Terraform provider supports multitenancy, allowing you to manage resources across multiple tenants.

When configuring the provider, make sure to specify the `tenant_id` parameter with the tenant ID you want to interact with.

```
provider "kestra" {
  tenant_id = "kestra-tech"
  url = "https://us.kestra.cloud"
}
```

Example configuration

::alert{type="warning"} Flows should always be deployed before Templates, to avoid flows running before their templates are created. ::

First, you need to create a `provider.tf` or `main.tf` file with the following content to configure the provider:

provider.tf

```
provider "kestra" {
  # mandatory, the URL for kestra
  url = "http://localhost:8080"

  # mandatory when using multitenancy, the ID of your tenant
  tenant_id = "kestra-tech"

  # optional basic auth username
```

```
  username = "john"

  # optional basic auth password
  password = "my-password"

  # optional API token, can be used instead of basic auth in the Enterprise Edition
  api_token = "my-api-token"

  # optional jwt token
  jwt = "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6Iktlc3RyYX
}
```

Then, you define the source and the version of the provider:

version.tf

```
kestra = {
  source  = "kestra-io/kestra"
  version = "~> 0.18.1"
}
```

Finally, you can create your resources:

flows.tf

```
resource "kestra_flow" "flow-example" {
  namespace = "company.team"
  flow_id = "myflow"
  content = file("kestra/flows/my-flow.yml")
}
```

templates.tf

```
resource "kestra_template" "template-example" {
  namespace = "company.team"
  template_id = "my-template"
  content = file("kestra/templates/my-template.yml")
  depends_on = [kestra_flow.flow-example] # Here we ensure that the flow is deployed before
}
```

More details can be found in the Terraform registry.

Develop a Trigger

Here is how you can develop a Trigger.

::collapse{title="The Trigger example below will create an execution randomly"}

```
@SuperBuilder
@ToString
@EqualsAndHashCode
@Getter
@NoArgsConstructor
public class Trigger extends AbstractTrigger implements PollingTriggerInterface, TriggerOutp
    @Builder.Default
    private final Duration interval = Duration.ofSeconds(60);

    protected Double min = 0.5;

    @Override
    public Optional<Execution> evaluate(ConditionContext conditionContext, TriggerContext co
        RunContext runContext = conditionContext.getRunContext();
        Logger logger = conditionContext.getRunContext().logger();
        double random = Math.random();

        if (random < this.min) {
            return Optional.empty();
        }

        Execution execution = Execution.builder()
            .id(runContext.getTriggerExecutionId())
            .namespace(context.getNamespace())
            .flowId(context.getFlowId())
            .flowRevision(context.getFlowRevision())
            .state(new State())
            .trigger(ExecutionTrigger.of(
                this,
                Trigger.Random.builder().random(random).build()
            ))
            .build();
```

```
        return Optional.of(execution);
    }

    @Builder
    @Getter
    public class Random implements io.kestra.core.models.tasks.Output {
        private Double random;
    }
}
```

::

You need to extend `PollingTriggerInterface` and implement the `Optional<Execution> evaluate(ConditionContext conditionContext, TriggerContext context)` method.

You can have any properties you want, like for any task (validation, documentation, …), and everything works the same way.

The `evaluate` method will receive these arguments: - `ConditionContext conditionContext`: a ConditionContext which includes various properties such as the RunContext in order to render your properties. - `TriggerContext context`: to have the context of this call (flow, execution, trigger, date, …).

In this method, you add any logic you want: connect to a database, connect to remote file systems, … You don't have to take care of resources, Kestra will run this method in its own thread.

This method must return an `Optional<Execution>` with: - `Optional.empty()`: if the condition is not validated. - `Optional.of(execution)`: with the execution created if the condition is validated.

You have to provide a `Output` for any output needed (result of query, result of file system listing, …) that will be available for the flow tasks within the `{{ trigger.* }}` variables.

::alert{type="warning"} Take care that the trigger must free the resource for the next evaluation. For each interval, this method will be called and if the conditions are met, an execution will be created.

To avoid this, move the file or remove the record from the database; take an action to avoid an infinite triggering. ::

Documentation

Remember to document your triggers. For this, we provide a set of annotations explained in the Document each plugin section.

Tutorial

Follow the tutorial to schedule and orchestrate your first workflows.

Kestra is an open-source orchestrator designed to bring Infrastructure as Code (IaC) best practices to all workflows — from those orchestrating mission-critical applications, IT operations, business processes, and data pipelines, to simple Zapier-style automations.

You can use Kestra to: - run workflows **on-demand**, **event-driven** or based on a regular **schedule** - programmatically interact with any system or programming language - orchestrate **microservices**, **batch jobs**, ad-hoc **scripts** (written in Python, R, Julia, Node.js, and more), **SQL queries**, data ingestion syncs, dbt or Spark jobs, or any other **applications** or **processes**

This tutorial will guide you through **key concepts** in Kestra. We'll build upon the "Hello world" flow from the Quickstart, and we'll gradually introduce new concepts including `namespaces`, `tasks`, parametrization with `inputs` and scheduling using `triggers`.

We'll then dive into `parallel` task execution, error handling, as well as custom scripts and microservices running in isolated containers. Let's get started!

::ChildCard{pageUrl="/docs/tutorial/"} ::

Users

Manage Users in Kestra

::alert{type=“info”} This feature requires a commercial license. ::

On the **Users** page, you will see the list of users.

By clicking on a user id or on the eye icon, you can open the page of a user.

The **Create** button allows creating a new user and managing that user's access to Kestra.

Users can be attached to Groups and/or Namespaces.

UI

Figure 1: UI

UI

Figure 2: UI

Webhook Trigger

Trigger flows based on web-based events.

Webhook triggers generates a unique URL that you can use to automatically create new executions based on events in another application such as GitHub or Amazon EventBridge.

In order to use that URL, you have to add a secret `key` that will secure your webhook URL.

```
type: "io.kestra.plugin.core.trigger.Webhook"
```

A Webhook trigger allows triggering a flow from a webhook URL. At trigger creation a key must be set that will be used on the URL that triggers the flow: `/api/v1/executions/webhook/{namespace}/{flowId}/{key}`. We advise to use a non-easy to find or remember key like a generated sequence of characters. Kestra accepts `GET`, `POST` and `PUT` requests on this URL The whole request body and headers will be available as variables.

Example

```
id: trigger
namespace: company.team

tasks:
  - id: hello
    type: io.kestra.plugin.core.log.Log
    message: "Hello World!  "

triggers:
  - id: webhook
    type: io.kestra.plugin.core.trigger.Webhook
    key: 4wjtkzwVGBM9yKnjm3yv8r
```

After the trigger is created, the key must be explicitly set in the webhook URL. You can execute the flow using the following URL:

`https://{kestra_domain}/api/v1/executions/webhook/{namespace}/{flowId}/4wjtkzwVGBM9yKnjm3yv8`

Make sure to replace `kestra_domain`, `namespace` and `flowId`.

Check the Webhook task documentation for the list of the task properties and outputs.

Manage Environments

Kestra users can manage their "environments" through different levels of granularity. Kestra has three main concepts: instance, tenant, and namespace.

When to use multiple instances?

An instance is a full deployment of Kestra. One best practice in Kestra is to have at least two separated instances: one for development and one for production.

The development serves as a "sandbox", only for development matters, while the production instance serves critical operations and is only accessible by administrators.

Large organizations sometimes have 3 or 4 environments: it's best to have Kestra Enterprise Edition to properly manage all these instances (thanks to improved governance, security and scalability provided by the Enterprise Edition of Kestra).

When to use multiple tenants?

Tenant are logic separations of an instance. You can think of them as isolated Kestra projects using instance resources. One instance can have many tenants.

This is useful when the operation managed by Kestra serves different customers or teams. For example, a company with 10 customers would use tenants to separate each of them in Kestra. We can imagine the same in an international company, using Kestra tenants country-wise.

One can also use tenants to separate engineer environments within the same development instance.

::alert{type="info"} Be aware that each tenant is using the same underlying instance resources. Therefore, it's not a best practice to use tenants to separate a development and a production environment. If the underlying instance is out of service for any reason, every tenant will be down too. ::

When to use multiple namespaces?

Namespaces are great for managing your flows organizations. One can use namespaces to arrange projects by domain or team.

They can be used as “environments” for getting started and for open-source users who don’t want to manage two or more instances. Again, it’s not a best practice for critical operations as one development can crash the flow supporting “production”.

Authentication

How to configure the authentication for your Kestra instance.

Kestra provides two authentication methods:

- Basic Auth: enabled by default
- OpenID Connect (OIDC)

By default, the JWT token security is configured to use the default Kestra encryption key. If you haven't already configured it, generate a secret that is at least 256 bits and add it to your kestra configuration as follows:

```yaml
kestra:
  encryption:
    secret-key: your-256-bits-secret
```

This secret must be the same on all your webserver instances and will be used to sign the JWT cookie and encode the refresh token.

If you want to use different keys, you can configure the key using the following configuration:

```yaml
micronaut:
  security:
    token:
      jwt:
        generator:
          refresh-token:
            secret: refresh-token-256-bits-secret
        signatures:
          secret:
            generator:
              secret: signature-256-bits-secret
```

::alert{type="info"} **JWT configuration**

It is possible to change the JWT cookie behavior using Micronaut Cookie Token Reader configuration. For example, you can define the cookie's maximum lifetime as `micronaut.security.token.cookie.cookie-max-age: P2D`. ::

Basic Authentification

The default installation comes with no users defined. To create an administrator account, use the following CLI command:

```
./kestra auths users create --admin --username=<admin-username> --password=<admin-password>
```

If you don't have multi-tenancy enabled, you can omit the `--tenant` parameter.

OpenID Connect (OIDC)

To enable OIDC in the application, make sure to enable OIDC in Micronaut:

```yaml
micronaut:
  security:
    oauth2:
      enabled: true
      clients:
        google:
          client-id: "{{ clientId }}"
          client-secret: "{{ clientSecret }}"
          openid:
            issuer: "{{ issuerUrl }}"
```

More information can be found in the Micronaut OIDC configuration.

Single Sign-On (SSO) with Google, Microsoft, and others

Check the Single Sign-On documentation for more details on how to configure SSO with Google, Microsoft, and other providers.

AWS Batch Task Runner

Run tasks as AWS ECS Fargate or EC2 containers using AWS Batch.

How to use the AWS Batch task runner

To launch tasks on AWS Batch, there are three main concepts you need to be aware of: 1. **Compute environment** — mandatory, it won't be created by the task. The compute environment is the infrastructure type for your tasks. It can be either an ECS Fargate or EC2 environment. 2. **Job Queue** — optional, it will be created by the task if not specified. Creating a queue takes some time to set up, so be aware that this adds some latency to the script's runtime. 3. **Job** — created by the task runner; holds information about which image, commands, and resources to run on. In AWS ECS terminology, it's the task definition for the ECS task.

::alert{type="info"} To get started quickly, follow the instructions from this blueprint to provision all resources required to run containers on ECS Fargate. ::

How does the AWS Batch task runner work?

In order to support `inputFiles`, `namespaceFiles`, and `outputFiles`, the AWS Batch task runner currently relies on multi-containers ECS jobs and creates three containers for each job: 1. A *before*-container that uploads input files to S3. 2. The *main* container that fetches input files into the `{{ workingDir }}` directory and runs the task. 3. An *after*-container that fetches output files using `outputFiles` to make them available from the Kestra UI for download and preview.

Since we don't know the working directory of the container in advance, we always need to explicitly define the working directory and output directory when using the AWS Batch runner, e.g. use `cat {{ workingDir }}/myFile.txt` rather than `cat myFile.txt`.

How to run tasks on AWS ECS Fargate

The example below shows how to use the AWS Batch task runner to offload the execution of Python scripts to a serverless container running on AWS ECS

Fargate:

```yaml
id: aws_batch_runner
namespace: company.team

tasks:
  - id: scrape_environment_info
    type: io.kestra.plugin.scripts.python.Script
    containerImage: ghcr.io/kestra-io/pydata:latest
    taskRunner:
      type: io.kestra.plugin.ee.aws.runner.Batch
      region: eu-central-1
      accessKeyId: "{{ secret('AWS_ACCESS_KEY_ID') }}"
      secretKeyId: "{{ secret('AWS_SECRET_KEY_ID') }}"
      computeEnvironmentArn: "arn:aws:batch:eu-central-1:707969873520:compute-environment/ke
      jobQueueArn: "arn:aws:batch:eu-central-1:707969873520:job-queue/kestraJobQueue"
      executionRoleArn: "arn:aws:iam::707969873520:role/kestraEcsTaskExecutionRole"
      taskRoleArn: arn:aws:iam::707969873520:role/ecsTaskRole
      bucket: kestra-product-de
    namespaceFiles:
      enabled: true
    outputFiles:
      - "*.json"
    script: |
      import platform
      import socket
      import sys
      import json
      from kestra import Kestra

      print("Hello from AWS Batch and kestra!")


      def print_environment_info():
          print(f"Host's network name: {platform.node()}")
          print(f"Python version: {platform.python_version()}")
          print(f"Platform information (instance type): {platform.platform()}")
          print(f"OS/Arch: {sys.platform}/{platform.machine()}")

          env_info = {
              "host": platform.node(),
              "platform": platform.platform(),
              "OS": sys.platform,
              "python_version": platform.python_version(),
          }
          Kestra.outputs(env_info)
```

```
        filename = "{{ workingDir }}/environment_info.json"
        with open(filename, "w") as json_file:
            json.dump(env_info, json_file, indent=4)


if __name__ == "__main__":
    print_environment_info()
```

::alert{type="info"} For a full list of properties available in the AWS Batch task runner, check the AWS plugin documentation or explore the same in the built-in Code Editor in the Kestra UI. ::

Full step-by-step guide: setting up AWS Batch from scratch

In order to use the AWS Batch task runner, you need to set up some resources in your AWS account. This guide will walk you through how you can configure the AWS Batch environment to run tasks on AWS ECS Fargate. We'll demonstrate two ways to set up the environment: 1. Using Terraform to provision all necessary resources using a simple `terraform apply` command. 2. Creating the resources step by step from the AWS Management Console.

Before you begin

Before you start, you need to have the following: 1. An AWS account. 2. Kestra Enterprise Edition instance in a version 0.18.0 or later with AWS credentials stored as secrets.

Terraform setup

Follow the instructions specified in the aws-batch/README within the terraform-deployments repository to provision all necessary resources using Terraform. You can also use the following blueprint that will create all necessary resources for you as part of a single Kestra workflow execution.

Here is a list of resources that will be created: - **AWS Security Group:** a security group for AWS Batch jobs with egress to the internet (required to be able to download public Docker images in your script tasks). - **AWS IAM Roles and Policies:** IAM roles and policies for AWS Batch and ECS Task Execution, including permissions for S3 access (S3 is used to store input and output files for container access). - **AWS Batch Compute Environment:** a managed ECS Fargate compute environment named `kestraFargateEnvironment`. - **AWS Batch Job Queue:** a job queue named `kestraJobQueue` for submitting batch jobs.

---

create_role

Figure 1: create_role

role_arn

Figure 2: role_arn

AWS Management Console setup

Create the `ecsTaskExecutionRole` IAM role To use AWS Batch, we need an Execution Role that will allow AWS Batch to create and manage resources on our behalf.

1. Open the IAM console.
2. In the navigation menu, choose **Roles**.
3. Choose **Create role**.
4. In the **Select trusted entity**, choose **Custom trust policy** and paste the following 'Trust policy JSON: `json { "Version": "2012-10-17", "Statement": [ { "Sid": "", "Effect": "Allow", "Principal": { "Service": "ecs-tasks.amazonaws.com" }, "Action": "sts:AssumeRole" } ] }` iam
5. Click on **Next** and add the `AmazonECSTaskExecutionRolePolicy`.
6. Then, for **Role Name**, enter `ecsTaskExecutionRole`
7. Finally, click on **Create role**.

Make sure to copy the ARN of the role. You will need it later.

Create the `ecsTaskRole` IAM role On top of the Execution Role, we will also need a Task Role that includes S3 access permissions to store files.

First, we'll need to create a policy the role can use for accessing S3.

1. Open the IAM console.
2. In the navigation menu, choose **Policies**.
3. Select **JSON** and paste the following into the `Policy editor`: `json { "Version": "2012-10-17", "Statement": [ { "Action": [ "s3:GetObject", "s3:PutObject", "s3:DeleteObject", "s3:ListBucket" ], "Effect": "Allow", "Resource": "*" } ] }` policy1
4. Select **Next** and type in a name for the policy, such as `ecsTaskRoleS3Policy`.
5. Once you're done, select **Create policy**.

policy2

Figure 3: policy2

batch4_search

Figure 4: batch4_search

batch4_firstrun

Figure 5: batch4_firstrun

Now we're ready to make our role. This is very similar to the previous role, but we need to add our new policy too:

1. Open the IAM console.
2. In the navigation menu, choose **Roles**.
3. Choose **Create role**.
4. In the **Select trusted entity**, choose **Custom trust policy** and paste the following 'Trust policy JSON: `json { "Version": "2012-10-17", "Statement": [ { "Sid": "", "Effect": "Allow", "Principal": { "Service": "ecs-tasks.amazonaws.com" }, "Action": "sts:AssumeRole" } ] }`
5. Click on **Next**
6. Search for the new policy and check the box on the left. Once you've done this, select **Next**. role_permission
7. Then, for **Role Name**, enter `ecsTaskRole`
8. Finally, click on **Create role**.

AWS Batch setup Go to the AWS Batch console.

Then, click on "Get Started". If you don't see the "Get Started" button, add `#firstRun` to the URL:

This will launch a Wizard that will guide you through the process of creating a new compute environment.

You should see the following text recommending the use of Fargate:

> "We recommend using Fargate in most scenarios. Fargate launches and scales the compute to closely match the resource requirements that you specify for the container. With Fargate, you don't need to over-provision or pay for additional servers. You also don't need to worry about the specifics of infrastructure-related parameters such as instance type. When the compute environment needs to be scaled up, jobs that run on Fargate resources can get started more quickly. Typically, it takes a few minutes to spin up a new Amazon EC2

batch4_jobtype

Figure 6: batch4_jobtype

batch5

Figure 7: batch5

batch6

Figure 8: batch6

> instance. However, jobs that run on Fargate can be provisioned in about 30 seconds. The exact time required depends on several factors, including container image size and number of jobs. Learn more."

We will follow that advice and use Fargate for this tutorial.

Step 1: Select Orchestration type Select Fargate and click on Next.

Step 2: Create a compute environment Add a name for your compute environment — here, we chose "kestra". You can keep the default settings for everything. Select the VPC and subnets you want to use — you can use the default VPC and subnets and the default VPC security group. Then, click on Next.

Step 3: Create a job queue Now we can create a job queue. Here, we also name it "kestra". You can keep the default settings. Then, click on Next:

Step 4: Create a job definition Finally, create a job definition. Here, we name it also "kestra". Under Execution role, select the role we created earlier (`ecsTaskExecutionRole`). Besides that, you can keep default settings for everything else (we adjusted the image to `ghcr.io/kestra-io/pydata:latest` but that's totally optional). Then, click on Next:

Step 5: Create a job Finally, create a job. Here, we name it "kestra". Then, click on Next for a final review:

Step 6: Review and create Review your settings and click on Create resources:

Once you see this message, you are all set:

batch7

Figure 9: batch7

batch8

Figure 10: batch8

batch9

Figure 11: batch9

Copy and apply the ARN to your Kestra configuration Copy the ARN of the compute environment and job queue. You will need to add these to your Kestra configuration.

Create an S3 Bucket

Last thing we'll need is an S3 storage bucket. To do this, select S3 from Services then select **Create bucket**.

Next you'll need to add a name and leave everything else as a default value.

Scroll to the bottom and select **Create bucket**.

Now that we have a bucket, we'll need to add the name into Kestra.

Run your Kestra task on AWS ECS Fargate

Fill in the ARNs of the compute environment and job queue in your Kestra configuration. Here is an example of a flow that uses the `aws.runner.Batch` to run a Python script on AWS ECS Fargate to get environment information and print it to the logs:

```
id: aws_batch_runner
namespace: company.team

variables:
  compute_environment_arn: arn:aws:batch:us-east-1:123456789:compute-environment/kestra
  job_queue_arn: arn:aws:batch:us-east-1:123456789:job-queue/kestra
  execution_role_arn: arn:aws:iam::123456789:role/ecsTaskExecutionRole
  task_role_arn: arn:aws:iam::123456789:role/ecsTaskRole

tasks:
  - id: send_data
    type: io.kestra.plugin.scripts.python.Script
    containerImage: ghcr.io/kestra-io/pydata:latest
    taskRunner:
```

batch10

Figure 12: batch10

batch11

Figure 13: batch11

batch12

Figure 14: batch12

```
  type: io.kestra.plugin.ee.aws.runner.Batch
  region: us-east-1
  accessKeyId: "{{ secret('AWS_ACCESS_KEY_ID') }}"
  secretKeyId: "{{ secret('AWS_SECRET_KEY_ID') }}"
  computeEnvironmentArn: "{{ vars.compute_environment_arn }}"
  jobQueueArn: "{{ vars.job_queue_arn}}"
  executionRoleArn: "{{ vars.execution_role_arn }}"
  taskRoleArn: "{{ vars.task_role_arn }}"
  bucket: kestra-us
script: |
  import platform
  import socket
  import sys

  print("Hello from AWS Batch and kestra!")

  def print_environment_info():
      print(f"Host's network name: {platform.node()}")
      print(f"Python version: {platform.python_version()}")
      print(f"Platform information (instance type): {platform.platform()}")
      print(f"OS/Arch: {sys.platform}/{platform.machine()}")

      try:
          hostname = socket.gethostname()
          ip_address = socket.gethostbyname(hostname)
          print(f"Host IP Address: {ip_address}")
      except socket.error as e:
          print("Unable to obtain IP address.")


  if __name__ == '__main__':
      print_environment_info()
```

When we execute this task, we can see the environment information inside of

s3_create

Figure 15: s3_create

s3_bucket_name

Figure 16: s3_bucket_name

logs

Figure 17: logs

the logs generated by the Python script:

Develop a Condition

Here is how you can develop a new Condition.

::collapse{title="Here is a simple condition example that validate the current flow:"}

```
@SuperBuilder
@ToString
@EqualsAndHashCode
@Getter
@NoArgsConstructor
@Schema(
    title = "Condition for a specific flow"
)
@Plugin(
    examples = {
        @Example(
            full = true,
            code = {
                "- conditions:",
                "    - type: io.kestra.plugin.core.condition.FlowCondition",
                "      namespace: company.team",
                "      flowId: my-current-flow"
            }
        )
    }
)
public class FlowCondition extends Condition {
    @NotNull
    @Schema(title = "The namespace of the flow")
    public String namespace;

    @NotNull
    @Schema(title = "The flow ID")
    public String flowId;

    @Override
    public boolean test(ConditionContext conditionContext) {
```

```
        return conditionContext.getFlow().getNamespace().equals(this.namespace) && condition
    }
}
```

::

You just need to extend `Condition` and implement the `boolean test(ConditionContext conditionContext)` method.

You can have any properties you want like for any task (validation, documentation, …), everything works the same way.

The `test` will receive a `ConditionContext` that will expose: - `conditionContext.getFlow()`: the current flow. - `conditionContext.getExecution()`: the current execution that can be null for Triggers. - `conditionContext.getRunContext()`: a RunContext in order to render your properties.

This method must simply return a boolean in order to validate the condition.

Documentation

Remember to document your conditions. For this, we provide a set of annotations explained in the Document each plugin section.

Building a Custom Docker Image

Build a custom Docker image for your script tasks.

You can bake all dependencies needed for your script tasks directly into the Kestra's base image. Here is an example installing Python dependencies:

```
FROM kestra/kestra:latest

USER root
RUN apt-get update -y && apt-get install pip -y

RUN pip install --no-cache-dir pandas requests boto3
```

Then, point to that Dockerfile in your `docker-compose.yml` file:

```
services:
  kestra:
    build:
      context: .
      dockerfile: Dockerfile
    image: kestra-python:latest
```

Once you start Kestra containers using `docker compose up -d`, you can create a flow that directly runs Python tasks with your custom dependencies using the `PROCESS` runner:

```
id: python_process
namespace: company.team
tasks:
  - id: custom_dependencies
    type: io.kestra.plugin.scripts.python.Script
    runner: PROCESS
    script: |
      import pandas as pd
      import requests
      import boto3
      print(f"Pandas version: {pd.__version__}")
      print(f"Requests version: {requests.__version__}")
      print(f"Boto3 version: {boto3.__version__}")
```

Building a custom Docker image for your script tasks

Imagine you use the following flow:

```yaml
id: zip_to_python
namespace: company.team

variables:
  file_id: "{{ execution.startDate | dateAdd(-3, 'MONTHS') | date('yyyyMM') }}"

tasks:
  - id: get_zipfile
    type: io.kestra.plugin.core.http.Download
    uri: "https://divvy-tripdata.s3.amazonaws.com/{{ render(vars.file_id) }}-divvy-tripdata

  - id: unzip
    type: io.kestra.plugin.compress.ArchiveDecompress
    algorithm: ZIP
    from: "{{ outputs.get_zipfile.uri }}"

  - id: parquet_output
    type: io.kestra.plugin.scripts.python.Script
    warningOnStdErr: false
    taskRunner:
      type: io.kestra.plugin.scripts.runner.docker.Docker
    containerImage: ghcr.io/kestra-io/pydata:latest
    env:
      FILE_ID: "{{ render(vars.file_id) }}"
    inputFiles: "{{ outputs.unzip.files }}"
    script: |
      import os
      import pandas as pd

      file_id = os.environ["FILE_ID"]
      file = f"{file_id}-divvy-tripdata.csv"

      df = pd.read_csv(file)
      df.to_parquet(f"{file_id}.parquet")
    outputFiles:
      - "*.parquet"
```

The Python task requires pandas to be installed. Pandas is a large library and it's not included in the default `python` image. In this case, you have the following options: 1. Install pandas in the `beforeCommands` property of the Python task. 2. Use one of our pre-built images that already include pandas, such as the `ghcr.io/kestra-io/pydata:latest` image. 3. Build your own custom Docker image that includes pandas.

1) Installing pandas in the `beforeCommands` property

```yaml
id: install_pandas_at_runtime
namespace: company.team
tasks:
  - id: custom_dependencies
    type: io.kestra.plugin.scripts.python.Script
    taskRunner:
      type: io.kestra.plugin.core.runner.Process
    beforeCommands:
      - pip install pyarrow pandas
    script: |
      import pandas as pd
      print(f"Pandas version: {pd.__version__}")
```

2) Using one of our pre-built images

```yaml
id: use_prebuilt_image
namespace: company.team
tasks:
  - id: custom_dependencies
    type: io.kestra.plugin.scripts.python.Script
    taskRunner:
      type: io.kestra.plugin.scripts.runner.docker.Docker
    containerImage: ghcr.io/kestra-io/pydata:latest
    script: |
      import pandas as pd
      print(f"Pandas version: {pd.__version__}")
```

3) Building a custom Docker image

If you want to build a custom Docker image for some of your scripts, first create a Dockerfile:

```
FROM python:3.11-slim
RUN pip install --upgrade pip
RUN pip install --no-cache-dir kestra requests pyarrow pandas amazon-ion
```

Then, build the image:

```
docker build -t kestra-custom:latest .
```

Finally, use that image in your flow:

```yaml
id: zip_to_python
namespace: company.team

variables:
  file_id: "{{ execution.startDate | dateAdd(-3, 'MONTHS') | date('yyyyMM') }}"
```

```yaml
tasks:
  - id: get_zipfile
    type: io.kestra.plugin.core.http.Download
    uri: "https://divvy-tripdata.s3.amazonaws.com/{{ render(vars.file_id) }}-divvy-tripdata.

  - id: unzip
    type: io.kestra.plugin.compress.ArchiveDecompress
    algorithm: ZIP
    from: "{{ outputs.get_zipfile.uri }}"

  - id: parquet_output
    type: io.kestra.plugin.scripts.python.Script
    warningOnStdErr: false
    taskRunner:
      type: io.kestra.plugin.scripts.runner.docker.Docker
      pullPolicy: NEVER #  Use the local image instead of pulling it from DockerHub
    containerImage: kestra-custom:latest #  Use your custom image here
    env:
      FILE_ID: "{{ render(vars.file_id) }}"
    inputFiles: "{{ outputs.unzip.files }}"
    script: |
      import os
      import pandas as pd

      file_id = os.environ["FILE_ID"]
      file = f"{file_id}-divvy-tripdata.csv"

      df = pd.read_csv(file)
      df.to_parquet(f"{file_id}.parquet")
    outputFiles:
      - "*.parquet"
```

Note how we use the `pullPolicy: NEVER` property to make sure that Kestra uses the local image instead of trying to pull it from DockerHub.

Executor

The **Executor** processes all executions and Flowable tasks.

The primary goal of the Executor is to receive created executions and look for the next tasks to run. This server component doesn't perform any heavy computation.

The Executor also handles special execution cases:

- Flow Triggers
- Templates (deprecated)
- Listeners (deprecated)

You can scale Executors as necessary. Given that no heavy computations are performed by this component, it requires very few resources (except for deployments with a large number of executions).

Version Control with Git

Setup Version Control with Git to store your flows and namespace files.

Kestra supports version control with Git. You can use one or more Git repositories to store your Flows and Namespace Files, and track changes to them over time via Git commit history.

There are multiple ways to use Git with Kestra: - The git.SyncFlows pattern allows you to implement GitOps and use Git as a single source of truth for your flows. - The git.SyncNamespaceFiles pattern allows you to implement GitOps and use Git as a single source of truth for your namespace files. - The git.PushFlows pattern allows you to edit your flows from the UI and regularly commit and push changes to Git; this pattern is useful if you want to use the built-in Editor in the UI and still have your code in Git. - The git.PushNamespaceFiles pattern allows you to edit your namespace files from the UI and regularly commit and push changes to Git; this pattern is useful if you want to use the built-in Editor in the UI and still have your files in Git. - The CI/CD pattern is useful if you want to manage the CI/CD process yourself e.g. via GitHub Actions or Terraform, and keep Git as a single source of truth for your code.

The image below shows how to choose the right pattern based on your needs:

Let's dive into each of these patterns, and when to use them.

Git SyncFlows and SyncNamespaceFiles

The Git SyncFlows pattern implements GitOps and uses Git as a single source of truth. It allows you to store your flows in Git and use a *system flow* that automatically syncs changes from Git to Kestra. You can also sync namespace files using the Git SyncNamespaceFiles pattern in the same way.

Here's how that works: - You store your flows and Namespace Files in Git -

git

Figure 1: git

You create a *system flow* that runs on a schedule and syncs changes from Git to Kestra - When you want to make a change to a flow or a namespace file, you modify the file in Git - The system flow syncs changes from Git to Kestra so that even if you make changes to any flows or Namespace Files from the UI, the changes are overwritten by the changes from Git.

This pattern is useful if you want to use Git as a single source of truth and avoid making changes to flows and Namespace Files from the UI. Using this pattern, you don't need to manage any CI/CD pipelines.

If your team follows the GitOps methodology, or you're coming from a Kubernetes background, this pattern is for you.

Here is an example system flow that you can use to declaratively sync changes from Git to Kestra:

```yaml
id: sync_from_git
namespace: system

tasks:
  - id: git
    type: io.kestra.plugin.git.SyncFlows
    url: https://github.com/kestra/scripts
    branch: main
    username: git_username
    password: "{{ secret('GITHUB_ACCESS_TOKEN') }}"
    targetNamespace: git
    includeChildNamespaces: true # optional; by default, it's set to false to allow explici
    gitDirectory: your_git_dir

triggers:
  - id: schedule
    type: io.kestra.plugin.core.trigger.Schedule
    cron: "*/1 * * * *" # every minute
```

You can choose to commit this flow to Git or add it from the built-in Editor in Kestra UI — this flow won't be overwritten by the Git reconciliation process.

You can also sync namespace files with the example below:

```yaml
id: sync_from_git
namespace: system


tasks:
  - id: git
    type: io.kestra.plugin.git.SyncNamespaceFiles
    namespace: prod
    gitDirectory: _files # optional; set to _files by default
```

github_webhook

Figure 2: github_webhook

```
    url: https://github.com/kestra-io/flows
    branch: main
    username: git_username
    password: "{{ secret('GITHUB_ACCESS_TOKEN') }}"
```

This flow can also be triggered anytime you push changes to Git via a GitHub webhook:

```
id: sync_from_git
namespace: system

tasks:
  - id: git
    type: io.kestra.plugin.git.SyncFlows
    url: https://github.com/kestra/scripts
    branch: main
    targetNamespace: git
    username: git_username
    password: "{{ secret('GITHUB_ACCESS_TOKEN') }}"

triggers:
  - id: github_webhook
    type: io.kestra.plugin.core.trigger.Webhook
    key: "{{ secret('WEBHOOK_KEY') }}"
```

Note that the webhook key is used to authenticate webhook requests and prevent unauthorized access to your Kestra instance. For the above flow, you would paste the following URL in your GitHub repository settings in the **Webhooks** section:

`https://us.kestra.cloud/api/v1/your_tenant/executions/webhook/prod/sync_from_git/your_secret`

Following the pattern:

`https://<host>/api/v1/<tenant>/executions/webhook/<namespace>/<flow>/<webhook_key>`

CI/CD

The CI/CD pattern allows you to use Git as a single source of truth and push code changes to Kestra anytime you merge a pull request. However, in contrast to the Git Sync pattern, you need to manage the CI/CD process yourself e.g. via GitHub Actions or Terraform. Check the CI/CD documentation for more details on how to set up CI/CD for Kestra flows and Namespace Files.

Git PushFlows and PushNamespaceFiles

The Git PushFlows pattern allows you to edit your flows from the UI, and regularly push changes to Git. It's particularly helpful if you want to use the built-in Editor in the UI and have your code change history managed via Git. You can also push namespace files using the Git PushNamespaceFiles pattern in the same way.

Here is example flow that you can use to push flow changes from Kestra to Git:

```yaml
id: push_to_git
namespace: system

tasks:
  - id: commit_and_push
    type: io.kestra.plugin.git.PushFlows
    url: https://github.com/kestra-io/scripts
    sourceNamespace: dev
    targetNamespace: pod
    flows: "*"
    branch: kestra
    username: github_username
    password: "{{ secret('GITHUB_ACCESS_TOKEN') }}"
    commitMessage: add namespace files changes

triggers:
  - id: schedule
    type: io.kestra.plugin.core.trigger.Schedule
    cron: "* */1 * * *" # every hour
```

Here is an example you can use to push namespace files from Kestra to Git:

```yaml
id: push_to_git
namespace: system

tasks:
  - id: commit_and_push
    type: io.kestra.plugin.git.PushNamespaceFiles
    namespace: dev
    files: "*"
    gitDirectory: _files
    url: https://github.com/kestra-io/scripts # required string
    username: git_username
    password: "{{ secret('GITHUB_ACCESS_TOKEN') }}"
    branch: dev
    commitMessage: "add namespace files"
```

```yaml
triggers:
  - id: schedule_push_to_git
    type: io.kestra.plugin.core.trigger.Schedule
    cron: "*/15 * * * *"
```

You can use that pattern to push changes to a feature branch and create a pull request for review. Once the pull request is approved, you can merge it to the main branch.

Git Clone

The Git Clone pattern allows you to clone a Git repository at runtime. This pattern can be used to orchestrate code maintained in a different code repository (potentially managed by a different team) in the following scenarios: - dbt projects orchestrated via dbt CLI task - infrastructure deployments orchestrated via Terraform CLI or Ansible CLI - Docker builds orchestrated via Docker Build task.

Helpers

Kestra provides some *helper* functions that can help during flow development.

::alert{type="warning"} These helpers are only available during flow development to test on your local installation. Before sending it to your server, you must expand the flow definition; our CI/CD support will automatically expand it. These helpers cannot be used from Kestra's UI. ::

Expand the flow to be uploaded to the server

There is a convenient command on the Kestra executable that allows validation of the current flow and will output the expanded version of your flow without any helper:

```
./kestra flow validate path-to-your-flow.yaml
```

`[[> file.txt]]`: Include another file

Working on a large flow can become complex when many tasks are defined, especially when you have some big text inside the flow definition (example, SQL statement, …).

Let's take an example:

```yaml
id: include
namespace: company.team

tasks:
- id: t1
  type: io.kestra.plugin.core.debug.Return
  format: |
    Lorem Ipsum is simply dummy text of the printing
    .....
    500 lines later
```

You can replace the flow definition with this one:

```yaml
id: include
namespace: company.team
```

```yaml
tasks:
- id: t1
  type: io.kestra.plugin.core.debug.Return
  format: "[[> lorem.txt]]"
```

And have a local file `lorem.txt` with the large content in it.

The path can be: * `[[> lorem.txt]]`: a relative path from the flow (flow.yaml and lorem.txt are in the same directory), * `[[> /path/to/lorem.txt]]`: an absolute path, * `[[> path/to/lorem.txt]]`: a relative path from the flow (flow.yaml with a subdirectory `path/to/`).

When including a file, you must use the right YAML scalar type: literal (with or without quotes) for single-line scalars or folded for multiple-lines ones.

::alert{type="warning"} Includes are resolved recursively, so you can include a file from another include. This allows more complex things, but you need to take care that included files don't contain `[[ .. ]]`. If you need to have these characters in included files, escape them with `\[[ ...]] !` ::

Validate the flow to be uploaded to the server

There is a convenient command on the Kestra executable that allows validation of the current flow:

```
./kestra flow validate --local path-to-your-flow.yaml
```

::alert{type="info"} If your flow uses a helper function, flow validation must be done locally as the flow cannot be expanded on the webserver. Be careful that the local installation must have the same plugins as the remote installation. ::

Expand the flow to be uploaded to the server

There is a convenient command on the Kestra executable that allows expanding of the current flow. It will resolve includes if any:

```
./kestra flow expand path-to-your-flow.yaml
```

Kubernetes on AWS EKS with Amazon RDS and S3

Deploy Kestra to AWS EKS with PostgreSQL RDS database and S3 internal storage backend.

Overview

This guide provides detailed instructions for deploying Kestra to AWS Elastic Kubernetes Service (EKS) with a PostgreSQL RDS database backend, and AWS S3 for internal storage.

Prerequisites: - Basic command line interface skills. - Familiarity with AWS EKS, RDS, S3, and Kubernetes.

Launch an EKS Cluster

First, install eksctl and kubectl. After installing them, you can create the EKS cluster. There are plenty of configuration options available with `eksctl`, but the default settings are sufficient for this guide. Run the following command to create a cluster named `my-kestra-cluster`:

```
eksctl create cluster --name my-kestra-cluster --region us-east-1
```

Wait for the cluster to be created. Then, confirm that the cluster is up, and that your kubecontext points to the cluster:

```
kubectl get svc
```

Launch AWS RDS for PostgreSQL

Navigate to the RDS console to create a PostgreSQL database. Configure the settings, ensuring the database is accessible from your EKS cluster. Note the database endpoint and port after creation.

Prepare an AWS S3 Bucket

Create a private S3 bucket (private meaning that public access is blocked). Keep a record of the bucket name as this is needed for the Kestra configuration.

Install Kestra on AWS EKS

Add the Kestra Helm chart repository and install Kestra:

```
helm repo add kestra https://helm.kestra.io/
helm install my-kestra kestra/kestra
```

In the deployment configuration, integrate RDS and S3. Set the database connection under `datasources` and S3 details under `storage` in your Helm values.

Here is how you can configure RDS in the Helm chart's values:

```
configuration:
  kestra:
    queue:
      type: postgres
    repository:
      type: postgres
  datasources:
    postgres:
      url: jdbc:postgresql://<your-rds-url-endpoint>:5432/kestra
      driverClassName: org.postgresql.Driver
      username: your_username
      password: your_password
```

Also, disable the PostgreSQL pod by changing the `enabled` value in the `postgresql` section from `true` to `false` in the same file.

```
postgresql:
  enabled: false
```

And here is how you can add the S3 configuration in the Helm chart's values:

```
configuration:
  kestra:
    storage:
      type: s3
      s3:
        accessKey: "<your-aws-access-key-id>"
        secretKey: "<your-aws-secret-access-key>"
        region: "<your-aws-region>"
        bucket: "<your-s3-bucket-name>"
```

Also, disable the Minio pod by changing the `enabled` value in the `minio` section from `true` to `false` in the same file.

```
minio:
  enabled: false
```

Apply these configurations using the following command:

```
helm upgrade kestra kestra/kestra -f values.yaml
```

Access Kestra UI

Implement an ingress controller for access. You can install the AWS Load Balancer (ALB) Controller via Helm:

```
helm install aws-load-balancer-controller eks/aws-load-balancer-controller \
-n kube-system \
--set clusterName=my-kestra-cluster \
--set serviceAccount.create=false \
--set serviceAccount.name=aws-load-balancer-controller
```

Once the ALB is set, you can access the Kestra UI through the ALB URL.

Next steps

This guide walked you through installing Kestra to AWS EKS with PostgreSQL RDS database and S3 storage backend.

Reach out via Slack if you encounter any issues or if you have any questions regarding deploying Kestra to production.

Polling Trigger

Trigger flows by polling external systems.

Polling triggers are a type of triggers that are provided by our plugins. They allow polling an external system for the presence of data. In case data is ready to be processed, a flow execution is started.

Kestra provides polling triggers for a wide variety of external systems, for example: databases, message brokers, ftp, …

Polling triggers will poll the external system at a fixed interval defined by the `interval` property, the triggered flow will have the outputs of the polling trigger available on the `trigger` variable.

Example

For example, the following flow will be triggered when rows are available on the `my_table` PostgreSQL table, and when triggered, it will delete the rows (to avoid processing them again on the next poll) and log them.

```yaml
id: jdbc-trigger
namespace: company.team

inputs:
  - id: db_url
    type: STRING

tasks:
- id: update
  type: io.kestra.plugin.jdbc.postgresql.Query
  url: "{{ inputs.db_url }}"
  sql: DELETE * FROM my_table

- id: log
  type: io.kestra.plugin.core.log.Log
  message: "{{ trigger.rows }}"

triggers:
  - id: watch
```

```
type: io.kestra.plugin.jdbc.postgresql.Trigger
url: myurl
interval: "PT5M"
sql: "SELECT * FROM my_table"
```

Polling triggers can be evaluated on a specific Worker Group (EE), thanks to the `workerGroup.key` property.

Secret

Store sensitive information securely.

Secret is a mechanism that allows you to securely store sensitive information, such as passwords and API keys, and retrieve them in your flows.

To retrieve secrets in a flow, use the `secret()` function, e.g. `"{{ secret('API_TOKEN'') }}"`. You can leverage your existing secrets manager as a secrets backend.

Your flows often need to interact with external systems. To do that, they need to programmatically authenticate using passwords or API keys. Secrets help you securely store such variables and avoid hard-coding sensitive information within your workflow code.

You can leverage the `secret()` function to retrieve sensitive variables within your flow code.

Secrets in the Enterprise Edition

Adding a new Secret from the UI

If you are using a managed Kestra version, you can add **new Secrets** directly from the UI. In the left navigation menu, go to **Namespaces**, select the namespace to which you want to add a new secret. Then, add a new secret within the Secrets tab.

Here, we add a new secret with a key `MY_SECRET`:

Using Secrets in your flows

For a concrete example of using secrets in flows, check out our dedicated How-To Guide on Secrets.

Secrets EE

Figure 1: Secrets EE

Secrets EE - new Secret

Figure 2: Secrets EE - new Secret

Secret Management backends

Kestra Enterprise Edition provides additional secret management backends and integrations with secrets managers. See the Secrets Manager page for more details.

Secrets in the Open-Source version

When using the open-source version, sensitive variables can be managed using base64-encoded environment variables. The section below demonstrates several ways to encode those values and use them in your Kestra instance.

Manual encoding using a CLI command

Imagine that so far, you were setting the following environment variable:

```
export MYPASSWORD=myPrivateCode
```

Here is how you can encode the sensitive value of that environment variable:

```
echo -n "myPrivateCode" | base64
```

This should output the value: `bXlQcml2YXRlQ29kZQ==`

To use that value as a Secret in your Kestra instance, you would need to add a prefix `SECRET_` to the variable key (here: `SECRET_MYPASSWORD`) and set that key to the encoded value:

```
export SECRET_MYPASSWORD=bXlQcml2YXRlQ29kZQ==
```

If you would add the environment variable to the `kestra` container section in a Docker Compose file, it would look as follows:

```
kestra:
  image: kestra/kestra:latest
  environment:
    SECRET_MYPASSWORD: bXlQcml2YXRlQ29kZQ==
```

This secret can then be used in a flow using the `{{ secret('MYPASSWORD') }}` syntax, and it will base64-decoded during flow execution. Make sure to not include the prefix `SECRET_` when calling the `secret('MYPASSWORD')` function, as this prefix is only there in the environment variable definition to prevent Kestra from treating other system variables as secrets (for better performance and increased security).

Lastly, shall you wish to reference any non_encoded environment variables in your flows definition, you can always use the syntax `{{envs.lowercase_environment_variable_key}}`.

::alert{type="warning"} Note that Kestra has built-in protection to prevent its logs from revealing any encoded secret you would have defined. ::

Convert all variables in an `.env` file

The previous section showed the process for one Secret. But what if you have tens or hundreds of them? This is where `.env` file can come in handy.

Let's assume that you have an `.env` file with the following content:

```
MYPASSWORD=password
GITHUB_ACCESS_TOKEN=mypat
AWS_ACCESS_KEY_ID=myawsaccesskey
AWS_SECRET_ACCESS_KEY=myawssecretaccesskey
```

Make sure to keep the last line empty, otherwise the bash script below won't encode the last secret AWS_SECRET_ACCESS_KEY correctly.

Using the bash script shown below, you can: 1. Encode all values using base64-encoding 2. Add a `SECRET_` prefix to all environment variable names 3. Store the result as `.env_encoded`

```
while IFS='=' read -r key value; do
    echo "SECRET_$key=$(echo -n "$value" | base64)";
done < .env > .env_encoded
```

The `.env_encoded` file should look as follows:

```
SECRET_MYPASSWORD=cGFzc3dvcmQ=
SECRET_GITHUB_ACCESS_TOKEN=bXlwYXQ=
SECRET_AWS_ACCESS_KEY_ID=bXlhd3NhY2Nlc3NrZXk=
SECRET_AWS_SECRET_ACCESS_KEY=bXlhd3NzZWNyZXRhY2Nlc3NrZXk=
```

Then, in your Docker Compose file, you can replace:

```
  kestra:
    image: kestra/kestra:latest
    env_file:
      - .env
```

with the encoded version of the file:

```
  kestra:
    image: kestra/kestra:latest
    env_file:
      - .env_encoded
```

Use a macro within your `.env` file

As an alternative to replacing values in your environment variables by encoded counterparts, you may also leverage the `base64encode` macro and keep the values intact.

The original `.env` file:

```
MYPASSWORD=password
GITHUB_ACCESS_TOKEN=mypat
AWS_ACCESS_KEY_ID=myawsaccesskey
AWS_SECRET_ACCESS_KEY=myawssecretaccesskey
```

can be modified to the following format:

```
SECRET_MYPASSWORD={{ "password" | base64encode }}
SECRET_GITHUB_ACCESS_TOKEN={{ "mypat" | base64encode }}
SECRET_AWS_ACCESS_KEY_ID={{ "myawsaccesskey" | base64encode }}
SECRET_AWS_SECRET_ACCESS_KEY={{ "myawssecretaccesskey" | base64encode }}
```

Service Accounts

Manage Service Accounts in Kestra.

::alert{type="info"} This feature requires a commercial license. ::

To create a new service account, go to the Service Accounts page under the Administration section and click on the **Create** button. Fill in the form with the required information including the name and description, and click **Save**:

Once you have created a service account, you can add a Role that will grant the service account permissions to specific resources. To do this, click on the **Add** button and select the role you want to assign to the service account.

Finally, you can generate an API token for the service account by clicking on the **Create** button. This will generate a token that you can use to authenticate the service account with Kestra from external applications such as CI/CD pipelines (e.g. in Terraform provider configuration or GitHub Actions secrets).

Note how you can configure the token to expire after a certain period of time, or to never expire. Also, there is a toggle called `Extended` that will automatically prolong the token's expiration date by the specified number of days (`Max Age`) if the token is actively used. That toggle is disabled by default.

Once you confirm the API token creation via the **Generate** button, the token will be generated and displayed in the UI. Make sure to copy the token and store it in a secure location as it will not be displayed again.

::alert{type="info"} Note that you can create an **API token** also as a regular **User**. While Service Accounts are generally recommended for programmatic API access to Kestra from CI/CD or other external applications, often it's useful to create an API token for a regular user, so that programmatic actions performed by that user can be tracked and audited. service_account_create_3 ::

service_account_create

Figure 1: service_account_create

service_account_create

Figure 2: service_account_create

service_account_create

Figure 3: service_account_create

service_account_create_2

Figure 4: service_account_create_2

Triggers

Triggers automatically start your flow based on events.

A trigger can be a scheduled date, a new file arrival, a new message in a queue, or the end of another flow's execution.

Defining triggers

Use the `triggers` keyword in the flow and define a list of triggers. You can have several triggers attached to a flow.

The `trigger` definition looks similar to the task definition — it contains an `id`, a `type`, and additional properties related to the specific trigger type.

The workflow below will be automatically triggered every day at 10 AM, as well as anytime when the `first_flow` finishes its execution. Both triggers are independent of each other.

```yaml
id: getting_started
namespace: company.team
tasks:
  - id: hello_world
    type: io.kestra.plugin.core.log.Log
    message: Hello World!

triggers:
  - id: schedule_trigger
    type: io.kestra.plugin.core.trigger.Schedule
    cron: 0 10 * * *

  - id: flow_trigger
    type: io.kestra.plugin.core.trigger.Flow
    conditions:
      - type: io.kestra.plugin.core.condition.ExecutionFlowCondition
        namespace: company.team
        flowId: first_flow
```

Add a trigger to your flow

Let's look at another trigger example. This trigger will start our flow every Monday at 10 AM.

```yaml
triggers:
  - id: every_monday_at_10_am
    type: io.kestra.plugin.core.trigger.Schedule
    cron: 0 10 * * 1
```

::collapse{title="Click here to see the full workflow example with this Schedule trigger"}

```yaml
id: getting_started
namespace: company.team

labels:
  owner: engineering

tasks:
  - id: api
    type: io.kestra.plugin.core.http.Request
    uri: https://dummyjson.com/products

  - id: python
    type: io.kestra.plugin.scripts.python.Script
    containerImage: python:slim
    beforeCommands:
      - pip install polars
    warningOnStdErr: false
    outputFiles:
      - "products.csv"
    script: |
      import polars as pl
      data = {{ outputs.api.body | jq('.products') | first }}
      df = pl.from_dicts(data)
      df.glimpse()
      df.select(["brand", "price"]).write_csv("products.csv")

  - id: sqlQuery
    type: io.kestra.plugin.jdbc.duckdb.Query
    inputFiles:
      in.csv: "{{ outputs.python.outputFiles['products.csv'] }}"
    sql: |
      SELECT brand, round(avg(price), 2) as avg_price
      FROM read_csv_auto('{{ workingDir }}/in.csv', header=True)
      GROUP BY brand
```

```
      ORDER BY avg_price DESC;
    store: true

triggers:
  - id: every_monday_at_10_am
    type: io.kestra.plugin.core.trigger.Schedule
    cron: 0 10 * * 1
```

::

To learn more about triggers, check out the full triggers documentation.

::next-link Next, let's orchestrate more complex workflows ::

Variables

Variables are key-value pairs that help reuse some values across tasks.

You can also store variables on a namespace level so that they can be reused across multiple flows in a given namespace.

How to configure variables

Here is how you can configure variables in your flow:

```yaml
id: hello_world
namespace: company.team

variables:
  myvar: hello
  numeric_variable: 42

tasks:
  - id: log
    type: io.kestra.plugin.core.debug.Return
    format: "{{ vars.myvar }} world {{ vars.numeric_variable }}"
```

You can see the syntax for using variables is `{{ vars.variable_name }}`.

How are variables rendered

You can use variables in any task property that is documented as **dynamic**.

Dynamic variables will be rendered thanks to the Pebble templating engine. Pebble templating engine allows you to process various expressions with filters and functions. More information on variable processing can be found under Expressions.

::alert{type="info"} Since 0.14, Variables are no longer rendered recursively. You can read more about this change and how to change this behaviour here. ::

Dynamic Variables

If you want to have an expression inside of your variable, you will need to wrap it in `render` when you use it in a task.

For example, this variable will only display the current time in the log message when wrapped in `render`. Otherwise, the log message will just contain the expression as a string:

```
id: dynamic_variable
namespace: company.team

variables:
  time: "{{ now() }}"

tasks:
  - id: log
    type: io.kestra.plugin.core.log.Log
    message: "{{ render(vars.time) }}"
```

::alert{type="info"} You will need to wrap the variable expression in `render` every time you want to use it in a task. ::

FAQ

How do I escape a block in Pebble syntax to ensure that it won't be parsed?

To ensure that a block of code won't be parsed by Pebble, you can use the `{% raw %}` and `{% endraw %}` Pebble tags. For example, the following Pebble expression will return the string `{{ myvar }}` instead of the value of the `myvar` variable:

```
{% raw %}{{ myvar }}{% endraw %}
```

Which order are inputs and variables resolved?

Inputs are resolved first, even before the execution starts. In fact, if you try to create a flow with an invalid input value, the execution will not be created.

Therefore, you can use inputs within variables, but you can't use variables or Pebble expressions within inputs.

Expressions are rendered recursively, meaning that if a variable contains another variable, the inner variable will be resolved first.

When it comes to triggers, they are handled similarly to inputs as they are known before the execution starts (they trigger the execution). This means that you can't use inputs (unless they have `defaults` attached) or variables within triggers, but you can use trigger variables within `variables`.

To make it clearer, let's look at some examples.

Examples This flow uses inputs, trigger and execution variables which are resolved before variables:

```yaml
id: upload_to_s3
namespace: company.team

inputs:
  - id: bucket
    type: STRING
    defaults: declarative-data-orchestration

tasks:
  - id: get_zip_file
    type: io.kestra.plugin.core.http.Download
    uri: https://wri-dataportal-prod.s3.amazonaws.com/manual/global_power_plant_database_v_1

  - id: unzip
    type: io.kestra.plugin.compress.ArchiveDecompress
    algorithm: ZIP
    from: "{{outputs.get_zip_file.uri}}"

  - id: csv_upload
    type: io.kestra.plugin.aws.s3.Upload
    from: "{{ outputs.unzip.files['global_power_plant_database.csv'] }}"
    bucket: "{{ inputs.bucket }}"
    key: "powerplant/{{ trigger.date ?? execution.startDate | date('yyyy_MM_dd__HH_mm_ss') }

triggers:
  - id: hourly
    type: io.kestra.plugin.core.trigger.Schedule
    cron: "@hourly"
```

This flow will start a task conditionally based on whether the input is provided or not:

```yaml
id: conditional_branching
namespace: company.team

inputs:
  - id: parameter
    type: STRING
    required: false

tasks:
  - id: if
    type: io.kestra.plugin.core.flow.If
    condition: "{{inputs.customInput ?? false }}"
    then:
      - id: if_not_null
        type: io.kestra.plugin.core.log.Log
```

```
        message: Received input {{inputs.parameter}}
    else:
      - id: if_null
        type: io.kestra.plugin.core.log.Log
        message: No input provided
```

Here is an example that uses a trigger variable within a trigger itself (*that's allowed!*):

```
id: backfill_past_mondays
namespace: company.team

tasks:
  - id: log_trigger_or_execution_date
    type: io.kestra.plugin.core.log.Log
    message: "{{ trigger.date ?? execution.startDate }}"

triggers:
  - id: first_monday_of_the_month
    type: io.kestra.plugin.core.trigger.Schedule
    timezone: Europe/Berlin
    backfill:
      start: 2023-11-11T00:00:00Z
    cron: "0 11 * * MON" # at 11 on every Monday
    conditions: # only first Monday of the month
      - type: io.kestra.plugin.core.condition.DayWeekInMonthCondition
        date: "{{ trigger.date }}"
        dayOfWeek: "MONDAY"
        dayInMonth: "FIRST"
```

Can I transform variables with Pebble expressions?

Yes. Kestra uses Pebble Templates along with the execution context to render **dynamic properties**. This means that you can use Pebble expressions (such as filters, functions, and operators to transform inputs and variables.

The example below illustrates how to use variables and Pebble expressions to transform string values in dynamic task properties:

```
id: variables_demo
namespace: company.team

variables:
  DATE_FORMAT: "yyyy-MM-dd"

tasks:
  - id: seconds_of_day
    type: io.kestra.plugin.core.debug.Return
```

```
    format: '{{60 * 60 * 24}}'

  - id: start_date
    type: io.kestra.plugin.core.debug.Return
    format: "{{ execution.startDate | date(vars.DATE_FORMAT) }}"

  - id: curr_date_unix
    type: io.kestra.plugin.core.debug.Return
    format: "{{ now() | date(vars.DATE_FORMAT) | timestamp() }}"

  - id: next_date
    type: io.kestra.plugin.core.debug.Return
    format: "{{ now() | dateAdd(1, 'DAYS') | date(vars.DATE_FORMAT) }}"

  - id: next_date_unix
    type: io.kestra.plugin.core.debug.Return
    format: "{{ now() | dateAdd(1, 'DAYS') | date(vars.DATE_FORMAT) | timestamp() }}"

  - id: pass_downstream
    type: io.kestra.plugin.scripts.shell.Commands
    taskRunner:
      type: io.kestra.plugin.core.runner.Process
    commands:
      - echo {{outputs.next_date_unix.value}}
```

Can I use nested variables?

Yes! However, keep in mind that depending on the task, you may need to wrap the root variable in a `json()` function in order to access specific keys. Here is an example using a list of maps as a variable:

```
id: vars
namespace: company.myteam

variables:
  servers:
    - fqn: server01.mydomain.io
      user: root
    - fqn: server02.mydomain.io
      user: guest
    - fqn: server03.mydomain.io
      user: rick

tasks:
  - id: parallel
    type: io.kestra.plugin.core.flow.EachParallel
```

```
value: "{{ vars.servers }}"
tasks:
  - id: log
    type: io.kestra.plugin.core.log.Log
    message:
       - "{{ taskrun.value }}" # for each element in the servers list, this will print
       - "{{ json(taskrun.value).fqn }}" # prints the value for that key e.g. server01.
       - "{{ json(taskrun.value).user }}" # prints the value for that key e.g. root
```

Workflow Components

Get to know the main orchestration components of a Kestra workflow.

::ChildCard{pageUrl="/docs/workflow-components/"} ::

Managing pip Package Dependencies

Learn how to manage pip package dependencies in your flows.

Motivation

Your Python code may require some `pip` package dependencies. The way you manage these dependencies can have an impact on the execution time of your flows.

If you install `pip` packages within `beforeCommands`, these packages will be downloaded and installed each time you run your task. This can lead to increased duration of your workflow executions. The following sections describe several ways to manage pip package dependencies in your flows.

Using a custom Docker image

Instead of using the Python Docker image, and installing pip package dependencies using `beforeCommands`, you can create a customer Docker image with Python and the required pip package dependencies. As all the pip packages would be part of this custom Docker image, you need not download and install the pip package dependencies during each execution. This would prevent the load on the execution, and the execution time will be dedicated to only the processing of the Python code.

For example, the Python example has `pandas` as a dependency. We can specify a Python container image that has this pre-installed, such as `ghcr.io/kestra-io/pydata:latest` meaning we don't need to use `beforeCommands`:

```yaml
id: docker_dependencies
namespace: company.team

tasks:
  - id: code
    type: io.kestra.plugin.scripts.python.Script
    taskRunner:
      type: io.kestra.plugin.scripts.runner.docker.Docker
    containerImage: ghcr.io/kestra-io/pydata:latest
```

```
  script: |
    import pandas as pd

    df = pd.read_csv('https://huggingface.co/datasets/kestra/datasets/raw/main/csv/orders.
    total_revenue = df['total'].sum()
```

Install pip package dependencies at server startup

This is another way of preventing the overload of downloading and installing pip package dependencies in each execution. You can install all the pip package dependencies, and then start the Kestra server. For Kestra standalone server, you can achieve this by running the command below:

```
pip install requests pandas polars && ./kestra server standalone --worker-thread=16
```

If you run Kestra using Docker, create a Dockerfile if you haven't already and install your dependencies using `RUN` inside of your Dockerfile. You will also need to set the `USER` to root for this to work.

```
FROM kestra/kestra:latest

USER root
RUN pip install requests pandas polars

CMD ["server", "standalone"]
```

Inside of your Docker Compose, you'll need to replace the `image` property with `build: .` to use our Dockerfile instead of the kestra image directly from Docker-Hub. Also, remove the `command` property as this is now handled in our Dockerfile with `CMD`:

```
services:
  ...
  kestra:
    build: .
    ...
```

When you run Kestra using Docker Compose, you will now see the Python dependencies added to the Dockerfile.

In either of these Kestra server installations, you will need to run the Python tasks using the Process Task Runner so that the Python code has access to the pip package dependencies installed in the Kestra server process.

We can check our dependencies are installed with the example below:

```
id: list_dependencies
namespace: company.team

tasks:
```

```
  - id: check
    type: io.kestra.plugin.scripts.python.Commands
    taskRunner:
      type: io.kestra.plugin.core.runner.Process
    commands:
      - pip list
```

Using cache files

In a `WorkingDirectory` task, you can have the virtual environment setup with the Process Task Runner, install all the pip package dependencies, and cache the `venv` folder. The pip package dependencies will then be cached as part of the virtual environment folder, and you need not install it on every execution of the flow. This is explained in detail in the caching page.

Here is a sample flow demonstrating how the `venv` folder can be cached:

```
id: python_cached_dependencies
namespace: company.team

tasks:
  - id: working_dir
    type: io.kestra.plugin.core.flow.WorkingDirectory
    tasks:
      - id: python_script
        type: io.kestra.plugin.scripts.python.Script
        taskRunner:
          type: io.kestra.plugin.core.runner.Process
        warningOnStdErr: false
        beforeCommands:
          - python -m venv venv
          - . venv/bin/activate
          - pip install pandas
        script: |
          import pandas as pd
          print(pd.__version__)
    cache:
      patterns:
        - venv/**
      ttl: PT24H
```

Thus, using one of the above techniques, you can prevent the installation of the pip package dependencies with every execution, and reduce your execution time.

Azure Batch Task Runner

Run tasks as containers on Azure Batch VMs.

How to use the Azure Batch task runner

This task runner will deploy a container for the task in a specified Azure Batch pool.

To launch the task on Azure Batch, there is only two main concepts you need to be aware of: 1. **Pool** — mandatory, not created by the task. This is a pool composed of nodes where your task can run on. 2. **Job** — created by the task runner; holds information about which image, commands, and resources to run on.

How does Azure Batch task runner work

In order to support `inputFiles`, `namespaceFiles`, and `outputFiles`, the Azure Batch task runner currently relies on resource files and output files which transit through Azure Blob Storage.

Since we don't know the working directory of the container in advance, we always need to explicitly define the working directory and output directory when using the Azure Batch runner, e.g. use cat `{{ workingDir }}/myFile.txt` rather than `cat myFile.txt`.

A full flow example

```yaml
id: azure_batch_runner
namespace: company.team

variables:
  poolId: "poolId"
  containerName: "containerName"

tasks:
  - id: scrape_environment_info
    type: io.kestra.plugin.scripts.python.Commands
    containerImage: ghcr.io/kestra-io/pydata:latest
```

```yaml
taskRunner:
  type: io.kestra.plugin.ee.azure.runner.Batch
  account: "{{ secret('AZURE_ACCOUNT') }}"
  accessKey: "{{ secret('AZURE_ACCESS_KEY') }}"
  endpoint: "{{ secret('AZURE_ENDPOINT') }}"
  poolId: "{{ vars.poolId }}"
  blobStorage:
    containerName: "{{ vars.containerName }}"
    connectionString: "{{ secret('AZURE_CONNECTION_STRING') }}"
commands:
  - python {{ workingDir }}/main.py
namespaceFiles:
  enabled: true
outputFiles:
  - "environment_info.json"
inputFiles:
  main.py: |
    import platform
    import socket
    import sys
    import json
    from kestra import Kestra

    print("Hello from Azure Batch and kestra!")

    def print_environment_info():
        print(f"Host's network name: {platform.node()}")
        print(f"Python version: {platform.python_version()}")
        print(f"Platform information (instance type): {platform.platform()}")
        print(f"OS/Arch: {sys.platform}/{platform.machine()}")

        env_info = {
            "host": platform.node(),
            "platform": platform.platform(),
            "OS": sys.platform,
            "python_version": platform.python_version(),
        }
        Kestra.outputs(env_info)

        filename = 'environment_info.json'
        with open(filename, 'w') as json_file:
            json.dump(env_info, json_file, indent=4)

    if __name__ == '__main__':
      print_environment_info()
```

search

Figure 1: search

create-account

Figure 2: create-account

::alert{type="info"} For a full list of properties available in the Azure Batch task runner, check the Azure plugin documentation or explore the same in the built-in Code Editor in the Kestra UI. ::

Full step-by-step guide: setting up Azure Batch from scratch

Before you begin

Before you start, you need to have the following: 1. An Microsoft Azure account. 2. A Kestra instance in a version 0.16.0 or later with Azure credentials stored as secrets or environment variables within the Kestra instance.

Azure Portal Setup

Create a Batch account and Azure Storage account Once you're logged into your Azure account, search for "Batch accounts" and select the first option under **Services**.

Now on that page, select **Create** to make a new account.

Select the appropriate resource group as well as fill in the Account name and Location fields. Afterwards, click on **Select a storage account**.

If you do not have an existing storage account, press **Create new** and type a name in, e.g. "mybatchstorage". Leave the other settings as the default options and select *OK*.

Now we have all the correct details filled, we can now press **Review + create** and then **Create** on the next page to create our new Batch account.

Once the account has been created, you'll receive a **Deployment succeeded** message appear. Select **Go to resource to go to the account.

Create a pool Now we have a Batch account, we can create a pool of compute notes in our Batch account that Kestra will use.

new-account

Figure 3: new-account

storage-account

Figure 4: storage-account

create-storage-account

Figure 5: create-storage-account

On the Batch account page, select **Pools** in the left navigation menu, then select **Add** at the top of the Pools page.

On the **Add pool** page, enter a Pool ID.

Under **Operating System**: - Select the **Publisher** as `microsoft-azure-batch` - Select the **Offer** as `ubuntu-server-container` - Select the **Sku** as `20-04-lts`

Scroll down to **Node size** and select **Standard_A1_v2** which is 1 vCPUs and 2 GB Memory. Also enter 2 for **Target dedicated nodes**.

Once you've done that, you can now select **OK** at the bottom to create the pool.

Create Access Key Inside of our Batch account, go to **Settings** and then **Keys**. Generate a new set of keys. We'll need: - `Batch account` for `account` - `Account endpoint` for `endpoint` - `Primary access key` for `accessKey`

Blob storage Search for Storage account and select our recently made account. Inside of here, go to the **Data storage** menu and select **Containers**. Now select + **Container** to make a new container.

Type in a name for the container and select **Create**.

Now that we've created our batch account, storage account, pool and container, we can now create our flow inside of Kestra.

Creating our Flow

Below is an example flow that will run a Python file called `main.py` on a Azure Batch Task Runner. At the top of the `io.kestra.plugin.scripts.python.Commands` task, there are the properties for defining our Task Runner:

```
containerImage: ghcr.io/kestra-io/pydata:latest
taskRunner:
  type: io.kestra.plugin.ee.azure.runner.Batch
  account: "{{ secret('AZURE_ACCOUNT') }}"
```

account-created

Figure 6: account-created

pools-menu

Figure 7: pools-menu

pool-name

Figure 8: pool-name

```
  accessKey: "{{ secret('AZURE_ACCESS_KEY') }}"
  endpoint: "{{ secret('AZURE_ENDPOINT') }}"
  poolId: "{{ vars.poolId }}"
  blobStorage:
    containerName: "{{ vars.containerName }}"
    connectionString: "{{ secret('AZURE_CONNECTION_STRING') }}"
```

This is where we can enter the details for Azure such as `account`, `accessKey`, `endpoint`, `poolId`, and `blobStorage`. We can add these as secrets and variables.

```
id: azure_batch_runner
namespace: company.team

variables:
  poolId: "poolId"
  containerName: "containerName"

tasks:
  - id: get_env_info
    type: io.kestra.plugin.scripts.python.Commands
    containerImage: ghcr.io/kestra-io/pydata:latest
    taskRunner:
      type: io.kestra.plugin.ee.azure.runner.Batch
      account: "{{ secret('AZURE_ACCOUNT') }}"
      accessKey: "{{ secret('AZURE_ACCESS_KEY') }}"
      endpoint: "{{ secret('AZURE_ENDPOINT') }}"
      poolId: "{{ vars.poolId }}"
      blobStorage:
        containerName: "{{ vars.containerName }}"
        connectionString: "{{ secret('AZURE_CONNECTION_STRING') }}"
    commands:
      - python {{ workingDir }}/main.py
    namespaceFiles:
      enabled: true
```

os

Figure 9: os

node-size

Figure 10: node-size

data-storage

Figure 11: data-storage

```
outputFiles:
  - "environment_info.json"
inputFiles:
  main.py: |
    import platform
    import socket
    import sys
    import json
    from kestra import Kestra

    print("Hello from Azure Batch and kestra!")

    def print_environment_info():
        print(f"Host's network name: {platform.node()}")
        print(f"Python version: {platform.python_version()}")
        print(f"Platform information (instance type): {platform.platform()}")
        print(f"OS/Arch: {sys.platform}/{platform.machine()}")

        env_info = {
            "host": platform.node(),
            "platform": platform.platform(),
            "OS": sys.platform,
            "python_version": platform.python_version(),
        }
        Kestra.outputs(env_info)

        filename = 'environment_info.json'
        with open(filename, 'w') as json_file:
            json.dump(env_info, json_file, indent=4)

    if __name__ == '__main__':
      print_environment_info()
```

When we press execute, we can see that our task runner is created in the Logs.

create-container

Figure 12: create-container

logs

Figure 13: logs

batch-jobs

Figure 14: batch-jobs

We can also go to the Azure Portal to see our task runner has been created:

Once the task has completed, it will automatically close down the runner on Azure.

We can also view the outputs generated in the Outputs tab in Kestra, which contains information about the Azure Batch task runner generated from our Python script:

outputs

Figure 15: outputs

Blueprints

Ready-to-use examples designed to kickstart your workflow.

Blueprints are a curated, organized, and searchable catalog of ready-to-use examples designed to help you kickstart your workflow. Each Blueprint combines code and documentation, and can be assigned several tags for organization and discoverability.

All Blueprints are validated and documented. You can easily customize and integrate them into your new or existing flows with a single click on the "Use" button.

Custom Blueprints

You can also create custom blueprints, shared within your organization.

::alert{type="info"} Custom blueprints require a commercial license. ::

Check the Blueprints documentation for more details.

Figure 1: Blueprint

Custom Blueprints

Figure 2: Custom Blueprints

Concepts

Learn the concepts and best practices to get the most out of Kestra.

::ChildCard{pageUrl="/docs/concepts/"} ::

Flowable Tasks

Run tasks or subflows in parallel, create loops and conditional branching.

Add parallelism using Flowable tasks

One of the most common orchestration requirements is to execute independent processes **in parallel**. For example, you can process data for each partition in parallel. This can significantly speed up the processing time.

The flow below uses the `EachParallel` flowable task to execute a list of `tasks` in parallel. 1. The `value` property defines the list of items to iterate over. 2. The `tasks` property defines the list of tasks to execute for each item in the list. You can access the iteration value using the `{{ taskrun.value }}` variable.

```yaml
id: python_partitions
namespace: company.team

description: Process partitions in parallel

tasks:
  - id: getPartitions
    type: io.kestra.plugin.scripts.python.Script
    taskRunner:
      type: io.kestra.plugin.scripts.runner.docker.Docker
    containerImage: ghcr.io/kestra-io/pydata:latest
    script: |
      from kestra import Kestra
      partitions = [f"file_{nr}.parquet" for nr in range(1, 10)]
      Kestra.outputs({'partitions': partitions})

  - id: processPartitions
    type: io.kestra.plugin.core.flow.EachParallel
    value: '{{ outputs.getPartitions.vars.partitions }}'
    tasks:
      - id: partition
        type: io.kestra.plugin.scripts.python.Script
        taskRunner:
          type: io.kestra.plugin.scripts.runner.docker.Docker
```

```
containerImage: ghcr.io/kestra-io/pydata:latest
script: |
  import random
  import time
  from kestra import Kestra

  filename = '{{ taskrun.value }}'
  print(f"Reading and processing partition {filename}")
  nr_rows = random.randint(1, 1000)
  processing_time = random.randint(1, 20)
  time.sleep(processing_time)
  Kestra.counter('nr_rows', nr_rows, {'partition': filename})
  Kestra.timer('processing_time', processing_time, {'partition': filename})
```

To learn more about flowable tasks, check out the full documentation.

::next-link Next, let's configure failure notifications and retries ::

Groups

Manage Groups in Kestra.

::alert{type="info"} This feature requires a commercial license. ::

On the **Groups** page, you will see the list of groups.

By clicking on a group id or on the eye icon, you can open the page of a group.

The **Create** button allows creating a new group and managing its access to Kestra.

It's a collection of users who require the same set of permissions. It's useful to assign the same permissions to multiple users who belong to the same team or project.

UI

Figure 1: UI

UI

Figure 2: UI

Inputs

Inputs is a list of dynamic values passed to the flow at runtime.

What are inputs

A flow can be parameterized using inputs to allow multiple executions of the same flow, each with different input values. Flow inputs are stored as variables within the flow execution context and can be accessed within the flow using the syntax `{{ inputs.parameter_name }}`.

You can use inputs to make your tasks more dynamic. For instance, you can use an input to dynamically define the path of a file that needs to be processed within a flow.

You can inspect the input values in the **Overview** tab of the **Execution** page.

You can set a custom `displayName` for each input to make the Execution interface more user-friendly.

Declaring inputs

You can declare as many inputs as necessary for any flow. Inputs can be **required** or **optional**.

If an input is required, we recommend using the `defaults` property to set default values. The flow cannot start if the input is not provided during the creation of the Execution.

Every input will be parsed during the creation of the execution, and any invalid inputs will deny the creation of the execution.

::alert{type="warning"} If the execution is **not created** due to invalid or missing inputs, no execution will be found on the list of executions. ::

Below is an example flow using several inputs:

```yaml
id: inputs
namespace: company.team

inputs:
  - id: string
```

```yaml
  type: STRING
  defaults: "Hello World!"
  displayName: "A string input"

- id: optional
  type: STRING
  required: false
  displayName: "An optional string"

- id: int
  type: INT
  defaults: 100
  displayName: "An integer input"

- id: list_of_int
  type: ARRAY
  itemType: INT
  defaults: [1, 2, 3]
  displayName: "A list of integers"

- id: bool
  type: BOOLEAN
  defaults: true
  displayName: "A boolean input"

- id: float
  type: FLOAT
  defaults: 100.12
  displayName: "A float input"

- id: dropdown
  type: SELECT
  displayName: "A dropdown input"
  defaults: VALUE_1
  values:
    - VALUE_1
    - VALUE_2
    - VALUE_3

- id: dropdown_multi
  type: MULTISELECT
  values:
    - VALUE_1
    - VALUE_2
    - VALUE_3
```

```yaml
- id: instant
  type: DATETIME
  defaults: "2013-08-09T14:19:00Z"
  displayName: "A datetime input"

- id: date
  type: DATE
  defaults: "2013-10-25"
  displayName: "A date input"

- id: time
  type: TIME
  displayName: "A time input"
  defaults: "14:19:00"

- id: duration
  type: DURATION
  defaults: "PT5M6S"
  displayName: "A duration input"

- id: file
  type: FILE
  displayName: "Upload a file"

- id: json
  type: JSON
  displayName: "A JSON input"
  defaults: |
    [{"name": "kestra", "rating": "best in class"}]

- id: uri
  type: URI
  defaults: "https://huggingface.co/datasets/kestra/datasets/raw/main/csv/orders.csv"
  displayName: "A URI input"

- id: secret
  type: SECRET
  displayName: "A secret input"

- id: yaml
  type: YAML
  defaults:
    - user: john
      email: john@example.com
    - user: will
      email: will@example.com
```

```
    displayName: YAML

  - id: nested.string
    type: STRING
    defaults: "Hello World!"
    displayName: "A nested string input"
```

::alert{type="info"} **Note:** `FILE` type does not support defaults currently. ::

Input types

Inputs in Kestra are strongly typed and validated before starting the flow execution.

Here is the list of supported data types:

- `STRING`: It can be any string value — inputs of type STRING are passed to the execution in its raw format without parsing; for additional validation, you can add a custom regex `validator` to allow only specific string patterns.
- `INT`: Must be a valid integer value (without any decimals).
- `FLOAT`: Must be a valid float value (with decimals).
- `SELECT`: Must be a valid string value from a predefined list of values. You can either pass those values directly using the `values` property or use the `expression` property to fetch the values dynamically from a KV store. Additionally, if `allowCustomValue` is set to true, the user can provide a custom value that is not in the predefined list.
- `MULTISELECT`: Must be one or more valid string values from a predefined list of values. You can either pass those values directly using the `values` property or use the `expression` property to fetch the values dynamically from a KV store. Additionally, if `allowCustomValue` is set to true, the user can provide a custom value that is not in the predefined list.
- `BOOLEAN`: Must be `true` or `false` passed as strings.
- `DATETIME`: Must be a valid full ISO 8601 date and time with the timezone expressed in UTC format; pass input of type DATETIME in a string format following the pattern `2042-04-02T04:20:42.00Z`.
- `DATE`: Must be a valid full ISO 8601 date without the timezone from a text string such as `2042-12-03`.
- `TIME`: Must be a valid full ISO 8601 time without the timezone from a text string such as `10:15:30`.
- `DURATION`: Must be a valid full ISO 8601 duration from a text string such as `PT5M6S`.
- `FILE`: Must be a file sent as `Content-Type: multipart/form-data` with `Content-Disposition: form-data; name="files"; filename="my-file"`, where `my-file` is the name of the input.
- `JSON`: Must be a valid JSON string and will be converted to a typed form.
- `YAML`: Must be a valid YAML string.

- `URI`: Must be a valid URI and will be kept as a string.
- `SECRET`: a `SECRET` input is a string that is encrypted and stored in the database. It is decrypted at runtime and can be used in all tasks. The value of a `SECRET` input is masked in the UI and in the execution context. Note that you need to set the encryption key in your Kestra configuration before using it.
- `ARRAY`: Must be a valid JSON array or a YAML list. The `itemType` property is required to ensure validation of the type of the array items.

All inputs of type `FILE` will be automatically uploaded to Kestra's internal storage and accessible to all tasks. After the upload, the input variable will contain a fully qualified URL of the form `kestra:///.../.../` that will be automatically managed by Kestra and can be used as-is within any task.

Input Properties

Below is the list of available properties for all inputs regardless of their types:

- `id`: The input parameter identifier — this property is important as it's used to reference the input variables in your flow, e.g. `{{ inputs.user }}` will reference the input parameter named `user`.
- `type`: The data type of the input parameter, as described in the previous section.
- `required`: Whether the input is required or optional; if required is set to true and no default value is configured and also no input is provided at runtime, the execution will not be created as Kestra cannot know what value to use.
- `defaults`: The default value that will be used if no custom input value is provided at runtime; this value must be provided as a string and will be coerced to the desired data type specified using the `type` property.
- `dependsOn`: Allows you to make the input dependent on another input to be inputted first.
- `displayName`: A different name that will display in the Execution interface instead of the `id`.
- `description`: A markdown description to document the input.

Input validation

Kestra validates the `type` of each input. In addition to the type validation, some input types can be configured with validation rules that will be enforced at execution time.

- `STRING`: A `validator` property allows the addition of a validation regex.
- `INT`: `min` and `max` properties allow the addition of minimum and maximum value ranges.
- `FLOAT`: `min` and `max` properties allow the addition of minimum and maximum ranges.

- `DURATION`: `min` and `max` properties allow the addition of minimum and maximum ranges.
- `DATE`: `after` and `before` properties help you ensure that the input value is within the allowed date range.
- `TIME`: `after` and `before` properties help you ensure that the input value is within the allowed time range.
- `DATETIME`: `after` and `before` properties help you ensure that the input value is within the allowed date range.

Example: using input validators in your flows

To ensure that your input value is within a certain `integer` value range, you can use the `min` and `max` properties. Similarly, to ensure that your string input matches a regex pattern, you can provide a custom regex `validator`. The following flow demonstrates how this can be accomplished:

```yaml
id: regex_input
namespace: company.team

inputs:
  - id: age
    type: INT
    defaults: 42
    required: false
    min: 18
    max: 64

  - id: user
    type: STRING
    defaults: student
    required: false
    validator: ^student(\d+)?$

  - id: float
    type: FLOAT
    defaults: 3.2
    min: 0.2
    max: 5.3

  - id: duration
    type: DURATION
    min: "PT5M6S"
    max: "PT12H58M46S"

  - id: date
    type: DATE
```

```
    defaults: "2024-04-12"
    after: "2024-04-10"
    before: "2024-04-15"

  - id: time
    type: TIME
    after: "11:01:01"
    before: "11:04:01"

  - id: datetime
    type: DATETIME
    defaults: "2024-04-13T14:17:00Z"
    after: "2024-04-10T14:19:00Z"
    before: "2024-04-15T14:19:00Z"

tasks:
  - id: validator
    type: io.kestra.plugin.core.log.Log
    message: User {{ inputs.user }}, age {{ inputs.age }}
```

The `age`, `float`, and `duration` input must be within a valid range between `min` and `max` values. Specifically for the `age` input, we specify that this input will be by default set to 42, but it can be overwritten at runtime to a value between 18 and 64. If you attempt to execute the flow with the `age` input set to 17 or 65, the validation will fail and the execution won't start.

Similarly, the Regex expression `^student(\d+)?$` is used to validate that the input argument `user` of type STRING follows the following pattern: - `^student`: This part of the regex asserts that the string must begin with the lowercase string value `student`. - `\d`: This part of the regex matches any digit (0-9). - `+`: This part of the regex asserts that there is one or more of the preceding token (i.e., one or more digits are allowed after the value `student`). - `()?`: The parentheses group the digits together, and the question mark makes the entire group optional — this means that the digits after the word `student` are optional. - `$`: This part of the regex asserts the end of the string. This ensures that the string doesn't contain any additional characters after the optional digits.

With this pattern: - "student" would be a match. - "student123" would be a match. - "studentabc" would not be a match because "abc" isn't a sequence of digits. - "student123abc" would not be a match because no more characters are allowed after the sequence of "students" with the following numbers.

Lastly, the `date`, `time`, and `datetime` inputs must be within a valid range between `after` and `before`. In the `date` example, the date provided must be between 10th May 2024 and 15th May 2024. Anything outside of this range will fail and the execution won't start.

Try running this flow with various inputs or adjust the regex pattern to see how

the input validation works.

Nested Inputs

If you use a `.` inside the name of an input, the input will be nested.

Here's an example that includes 2 nested inputs:

```yaml
id: nested_inputs
namespace: company.team

inputs:
  - id: nested.string
    type: STRING
    required: false

  - id: nested.int
    type: INT

tasks:
  - id: log_inputs
    type: io.kestra.plugin.core.log.Log
    message: "{{ inputs.nested.string }} and {{ inputs.nested.int }}"
```

You can access the first input value using `{{ inputs.nested.string }}`. This syntax provides a convenient type validation of nested inputs without using raw JSON that would not be validated (JSON-type input values are passed as strings).

Array Inputs

Array inputs are used to pass a list of values to a flow. The `itemType` property is required to ensure validation of the type of the array items.

It's particularly useful when you want the end-user triggering the workflow to provide multiple values of a specific type, e.g. a list of integers, strings, booleans, datetimes, etc. You can provide the default values as a JSON array or as a YAML list — both are supported.

```yaml
id: array_demo
namespace: company.team

inputs:
  - id: my_numbers_json_list
    type: ARRAY
    itemType: INT
    defaults: [1, 2, 3]
```

array_inputs

Figure 1: array_inputs

```
  - id: my_numbers_yaml_list
    type: ARRAY
    itemType: INT
    defaults:
      - 1
      - 2
      - 3

tasks:
  - id: print_status
    type: io.kestra.plugin.core.log.Log
    message: received inputs {{ inputs }}
```

Here is how the array inputs are rendered in the UI when you create an execution:

Using an input value in a flow

Every input is available with dynamic variables such as: `{{ inputs.name }}` or `{{ inputs['name'] }}`. If you use characters inside of your input id such as `-`, you'll need to use the `{{ inputs['name-example']}}` format.

For example, if you declare the following inputs:

```
inputs:
  - id: mystring
    type: STRING
    required: true

  - id: my-file
    type: FILE
```

You can use the value of the input `mystring` inside dynamic task properties with `{{ inputs.mystring }}` but `my-file` would have to use `{{ inputs['my-file'] }}` because of the `-`.

We can see a full example here where `inputFiles` property is set to `{{ inputs['my-file'] }}`:

```
id: input_files
namespace: company.team

description: This flow shows how to pass files between inputs and tasks in Shell scripts.
```

```yaml
inputs:
  - id: my-file
    type: FILE

tasks:
  - id: rename
    type: io.kestra.plugin.scripts.shell.Commands
    commands:
      - mv file.tmp output.tmp
    inputFiles:
      file.tmp: "{{ inputs['my-file'] }}"
    outputFiles:
      - "*.tmp"
```

::alert{type="info"} Since 0.14, Inputs are no longer rendered recursively. You can read more about this change and how to change this behaviour here. ::

Set input values at flow execution

When you execute a flow with inputs, you must set all inputs (unless optional or with a default value) to be able to create the execution.

Let's consider the following example that defines multiple inputs:

```yaml
id: kestra_inputs
namespace: company.team

inputs:
  - id: string
    type: STRING
    defaults: hello

  - id: optional
    type: STRING
    required: false

  - id: int
    type: INT

  - id: float
    type: FLOAT

  - id: instant
    type: DATETIME

  - id: file
    type: FILE
```

Flow inputs

Figure 2: Flow inputs

Here, the inputs `{{ inputs.string }}` and `{{ inputs.optional }}` can be skipped because the `string` input has a default value and the `optional` input is not required. All other inputs must be specified at runtime.

Set inputs from the web UI

When creating an execution from the web UI, you must set the inputs in the UI form. Kestra's UI will generate a dedicated form based on your `inputs` definition. For example, datetime input properties will have a date picker.

The input form for the inputs above looks as follows:

Once the inputs are set, you can trigger an execution of the flow.

Set inputs when executing the flow using the API

To create an execution with these inputs using the API, we can use the `curl` command to make an API request:

```
curl -v "http://localhost:8080/api/v1/executions/example/kestra-inputs" \
    -H "Transfer-Encoding:chunked" \
    -H "Content-Type:multipart/form-data" \
    -F string="a string"  \
    -F optional="an optional string"  \
    -F int=1  \
    -F float=1.255  \
    -F instant="2023-12-24T23:00:00.000Z" \
    -F "files=@/tmp/128M.txt;filename=file"
```

All files must be sent as multipart form data named `files` with a header `filename=my-file` which will be the name of the input.

Set inputs when executing the flow in Python

To create an execution with these inputs in Python, you can use the following script:

```
import io
import requests
from kestra import Flow

flow = Flow()

with open('/tmp/example.txt', 'rb') as fh:
  flow.execute('example',
```

```
                'kestra-inputs',
                {'string': 'a string',
                 'optional': 'an optional string',
                 'int': 1,
                 'float': str(1.255),
                 'instant': '2020-01-14T23:00:00.000Z',
                 'files': ('file', fh, 'text/plain')})
```

::alert{type="info"} Floats need to be wrapped in `str()` otherwise you will run into a bytes-like object error when passing a file as an input too. ::

You can also use the `requests` library in Python to make requests to the Kestra API. Here's an example to execute a flow with multiple inputs:

```python
import io
import requests
from requests_toolbelt.multipart.encoder import MultipartEncoder

with open("/tmp/128M.txt", 'rb') as fh:
  url = f"http://kestra:8080/api/v1/executions/io.kestra.docs/my-flow"
  mp_encoder = MultipartEncoder(fields={
    "string": "a string",
    "optional": "an optionnal string",
    "int": 1,
    "float": 1.255,
    "instant": "2020-01-14T23:00:00.000Z",
    "files": ("file", fh, "text/plain")
  })
  result = requests.post(
      url,
      data=mp_encoder,
      headers={"Content-Type": mp_encoder.content_type},
  )
```

Set inputs when executing the flow in Java

To create an execution with these inputs in Java (with Apache Http Client 5), you can use the following script:

```java
import org.apache.hc.client5.http.classic.methods.HttpPost;
import org.apache.hc.client5.http.entity.mime.FileBody;
import org.apache.hc.client5.http.entity.mime.MultipartEntityBuilder;
import org.apache.hc.client5.http.entity.mime.StringBody;
import org.apache.hc.client5.http.impl.classic.CloseableHttpClient;
import org.apache.hc.client5.http.impl.classic.CloseableHttpResponse;
import org.apache.hc.client5.http.impl.classic.HttpClientBuilder;
import org.apache.hc.core5.http.ContentType;
import org.apache.hc.core5.http.HttpEntity;
```

```
import java.io.File;

class Application {
  public static void main(String[] args) {
    HttpEntity multipartEntity = MultipartEntityBuilder.create()
        .addPart("string", new StringBody("test", ContentType.DEFAULT_TEXT))
        .addPart("int", new StringBody("1", ContentType.DEFAULT_TEXT))
        .addPart("files", new FileBody(new File("/tmp/test.csv"), ContentType.DEFAULT_TEXT,
        .build();

    try (CloseableHttpClient httpclient = HttpClientBuilder.create().build()) {
      HttpPost request = new HttpPost("http://kestra:8080/api/v1/executions/com.kestra.lde/i
      request.setEntity(multipartEntity);

      CloseableHttpResponse response = httpclient.execute(request);

      System.out.println("Response " + response);
    } catch (Exception e) {
      throw new RuntimeException(e);
    }
  }
}
```

Difference between inputs and variables

Variables are similar to constant values. They have the same behaviour as an input during execution but they can't be overridden once the execution starts. Variables must be defined before execution whereas inputs can be set at execution.

Variables are best suited for values that you don't want to change, and are used in multiple places within the flow. For example, a URL you will use for an API request that won't change would be best as a variable whereas an email address that changes every time you execute your flow would be best as an input.

Dynamic inputs

Inputs in kestra are strongly typed. Currently, it's not possible to enforce strong types and simultaneously use dynamically rendered Pebble expressions. However, you can use Pebble expressions in default values within STRING inputs.

This example wouldn't work:

```
id: test
namespace: company.team
```

```yaml
inputs:
  - id: date
    type: DATETIME
    defaults: "{{ now() }}"

tasks:
  - id: print_date
    type: io.kestra.plugin.core.log.Log
    message: hello on {{ inputs.date }}
```

However, if you change the input type to STRING, you can use Pebble expressions such as `{{ now() }}` in the default value:

```yaml
id: test
namespace: company.team

inputs:
  - id: date
    type: STRING
    defaults: "{{ now() }}"

tasks:
  - id: print_date
    type: io.kestra.plugin.core.log.Log
    message: hello on {{ render(inputs.date) }}
```

Keep in mind that since Kestra 0.14, inputs are no longer rendered recursively. Therefore, you need to use the `{{ render(inputs.date) }}` syntax to render the Pebble expression specified within the STRING input value. This improves security by preventing the execution of arbitrary code within the Pebble expression.

You can read more about this change in the Migration Guide.

Conditional Inputs for Interactive Workflows

Starting in Kestra 0.19.0, you can set up inputs that depend on other inputs, letting further inputs to be conditionally displayed based on user choices. This is useful for use cases such as approval workflows or dynamic resource provisioning.

How It Works

You can create inputs that change based on conditions using the `dependsOn` and `condition` properties. Here's an example where different inputs show up depending on the resource type a user selects:

```yaml
id: request_resources
namespace: company.team
```

```yaml
inputs:
  - id: resource_type
    displayName: Resource Type
    type: SELECT
    values:
      - Access permissions
      - SaaS application
      - Development tool
      - Cloud VM

  - id: access_permissions
    displayName: Access Permissions
    type: SELECT
    expression: "{{ kv('access_permissions') }}"
    dependsOn:
      inputs:
        - resource_type
      condition: "{{ inputs.resource_type == 'Access permissions' }}"

  - id: saas_applications
    displayName: SaaS Application
    type: MULTISELECT
    expression: "{{ kv('saas_applications') }}"
    dependsOn:
      inputs:
        - resource_type
      condition: "{{ inputs.resource_type == 'SaaS application' }}"

  - id: cloud_provider
    displayName: Cloud Provider
    type: SELECT
    values:
      - AWS
      - GCP
      - Azure
    dependsOn:
      inputs:
        - resource_type
      condition: "{{ inputs.resource_type == 'Cloud VM' }}"

  - id: cloud_vms
    displayName: Cloud VM
    type: SELECT
    expression: "{{ kv('cloud_vms')[inputs.cloud_provider] }}"
    dependsOn:
      inputs:
```

```
      - resource_type
      - cloud_provider
    condition: "{{ inputs.resource_type == 'Cloud VM' }}"
```

In this example: - The `resource_type` input controls which additional inputs (like `access_permissions`, `saas_applications`, and `cloud_vms`) appear. - The `dependsOn` property links the inputs, and the `condition` property defines when to display the related input.

Before running the above flow, you need to set up the key-value pairs for each input. Expand the example below to see how to set up the key-value pairs for all inputs using a flow.

::collapse{title="Flow adding key-value pairs"}

```
id: add_kv_pairs
namespace: company.team

tasks:
  - id: access_permissions
    type: io.kestra.plugin.core.kv.Set
    key: "{{ task.id }}"
    kvType: JSON
    value: |
      ["Admin", "Developer", "Editor", "Launcher", "Viewer"]

  - id: saas_applications
    type: io.kestra.plugin.core.kv.Set
    key: "{{ task.id }}"
    kvType: JSON
    value: |
      ["Slack", "Notion", "HubSpot", "GitHub", "Jira"]

  - id: development_tools
    type: io.kestra.plugin.core.kv.Set
    key: "{{ task.id }}"
    kvType: JSON
    value: |
      ["Cursor", "IntelliJ IDEA", "PyCharm Professional", "Datagrip"]

  - id: cloud_vms
    type: io.kestra.plugin.core.kv.Set
    key: "{{ task.id }}"
    kvType: JSON
    value: |
      {
        "AWS": ["t2.micro", "t2.small", "t2.medium", "t2.large"],
        "GCP": ["f1-micro", "g1-small", "n1-standard-1", "n1-standard-2"],
```

```
        "Azure": ["Standard_B1s", "Standard_B1ms", "Standard_B2s", "Standard_B2ms"]
      }

  - id: cloud_regions
    type: io.kestra.plugin.core.kv.Set
    key: "{{ task.id }}"
    kvType: JSON
    value: |
      {
        "AWS": ["us-east-1", "us-west-1", "us-west-2", "eu-west-1"],
        "GCP": ["us-central1", "us-east1", "us-west1", "europe-west1"],
        "Azure": ["eastus", "westus", "centralus", "northcentralus"]
      }
```

::

You could also add those key-value pairs using the API or from the UI.

Custom Values in the SELECT and MULTISELECT Inputs

If you want to allow users to enter other value if the predefined values shown in the dropdown don't fit their needs, you can set `allowCustomValue` to `true` on any input. This enables you to provide a list of default values but still (optionally) allow users to enter custom ones.

In the example below, the `cloud_provider` input allows users to select from a list of the most common cloud providers (AWS, GCP, Azure) or enter a custom value if they use some less common cloud provider e.g. Oracle Cloud.

```yaml
id: custom_values
namespace: company.team

inputs:
  - id: cloud_provider
    displayName: Cloud Provider
    type: SELECT
    allowCustomValue: true
    values:
      - AWS
      - GCP
      - Azure

tasks:
  - id: print_status
    type: io.kestra.plugin.core.log.Log
    message: selected cloud provider {{ inputs.cloud_provider }}
```

Installing Dependencies at Runtime

Install dependencies at runtime using `beforeCommands`.

There are several ways of installing custom packages for your workflows. This page shows how to install dependencies at runtime using the `beforeCommands` property.

Installing dependencies using `beforeCommands`

While you could bake all your package dependencies into a custom container image, often it's convenient to install a couple of additional packages at runtime without having to build separate images. The `beforeCommands` can be used for that purpose.

pip install package

Here is a simple example installing `pip` packages `requests` and `kestra` before starting the script:

```yaml
id: pip
namespace: company.team

tasks:
  - id: before_commands
    type: io.kestra.plugin.scripts.python.Script
    containerImage: python:3.11-slim
    beforeCommands:
      - pip install requests kestra > /dev/null
    script: |
      import requests
      import kestra

      kestra_modules = [i for i in dir(kestra.Kestra) if not i.startswith("_")]

      print(f"Requests version: {requests.__version__}")
      print(f"Kestra modules: {kestra_modules}")
```

pip install -r requirements.txt

This example clones a Git repository that contains a `requirements.txt` file. The script task uses `beforeCommands` to install those packages. We then list recently installed packages to validate that this process works as expected:

```yaml
id: python_requirements_file
namespace: company.team

tasks:
  - id: wdir
    type: io.kestra.plugin.core.flow.WorkingDirectory
    tasks:
      - id: cloneRepository
        type: io.kestra.plugin.git.Clone
        url: https://github.com/kestra-io/examples
        branch: main

      - id: print_requirements
        type: io.kestra.plugin.scripts.shell.Commands
        taskRunner:
          type: io.kestra.plugin.core.runner.Process
        commands:
          - cat requirements.txt

      - id: list_installed_packages
        type: io.kestra.plugin.scripts.python.Commands
        warningOnStdErr: false
        containerImage: python:3.11-slim
        beforeCommands:
          - pip install -r requirements.txt > /dev/null
        commands:
          - ls -lt $(python -c "import site; print(site.getsitepackages()[0])") | head -n 20
```

And here is a simple version where we add the `requirements.txt` file using the `inputFiles` property:

```yaml
id: python_requirements_file
namespace: company.team

tasks:
  - id: list_installed_packages
    type: io.kestra.plugin.scripts.python.Script
    env:
      PIP_ROOT_USER_ACTION: ignore
    inputFiles:
      requirements.txt: |
```

```
    polars
    requests
    kestra
containerImage: python:3.11-slim
beforeCommands:
  - pip install --upgrade pip
  - pip install -r requirements.txt > /dev/null
script: |
  from kestra import Kestra
  import pkg_resources
  import re

  with open('requirements.txt', 'r') as file:
      # find package names without versions
      required_packages = {re.match(r'^\s*([a-zA-Z0-9_-]+)', line).group(1) for line in

  installed_packages = [(d.project_name, d.version) for d in pkg_resources.working_set]

  kestra_outputs = {}

  for name, version in installed_packages:
      if name in required_packages:
          kestra_outputs[name] = version

  Kestra.outputs(kestra_outputs)
```

As you can see here, the `WorkingDirectory` task is usually only needed if you use the `git.Clone` task. In most other cases, you can use the `inputFiles` property to add files to the script's working directory.

Using Kestra's prebuilt images

Many data engineering use cases require performing fairly standardized tasks such as:

- processing data with `pandas`
- transforming data with `dbt-core` (*using a dbt adapter for your data warehouse*)
- making API calls with the `requests` library, etc.

To solve those common challenges, the kestra-io/examples repository provides several **public** Docker images with the latest versions of those common packages. Many Blueprints use those public images by default. The images are hosted in GitHub Container Registry managed by Kestra's team and those images follow the naming `ghcr.io/kestra-io/packageName:latest`.

Example: running R script in Docker

Here is a simple example using the `ghcr.io/kestra-io/rdata:latest` Docker image provided by Kestra to analyze the built-in `mtcars` dataset using `dplyr` and `arrow` R libraries:

```yaml
id: rCars
namespace: company.team

tasks:
  - id: r
    type: io.kestra.plugin.scripts.r.Script
    warningOnStdErr: false
    containerImage: ghcr.io/kestra-io/rdata:latest
    outputFiles:
      - "*.csv"
      - "*.parquet"
    script: |
      library(dplyr)
      library(arrow)
      data(mtcars) # Load mtcars data
      print(head(mtcars))
      final <- mtcars %>%
        summarise(
          avg_mpg = mean(mpg),
          avg_disp = mean(disp),
          avg_hp = mean(hp),
          avg_drat = mean(drat),
          avg_wt = mean(wt),
          avg_qsec = mean(qsec),
          avg_vs = mean(vs),
          avg_am = mean(am),
          avg_gear = mean(gear),
          avg_carb = mean(carb)
        )
      final %>% print()
      write.csv(final, "final.csv")
      mtcars_clean <- na.omit(mtcars) # remove rows with NA values
      write_parquet(mtcars_clean, "mtcars_clean.parquet")
```

Installation of R libraries is time-consuming. From a technical standpoint, you could install custom R packages at runtime as follows:

```yaml
id: rCars
namespace: company.team

tasks:
```

```yaml
- id: r
  type: io.kestra.plugin.scripts.r.Script
  warningOnStdErr: false
  containerImage: ghcr.io/kestra-io/rdata:latest
  beforeCommands:
    - Rscript -e "install.packages(c('dplyr', 'arrow'))" > /dev/null 2>&1
```

However, that flow above might take up to 30 minutes, depending on the R packages you install.

Prebuilt Docker images such as `ghcr.io/kestra-io/rdata:latest` can help you iterate much faster. Before moving to production, you can build your custom images with the exact package versions that you need.

Kubernetes on GCP GKE with CloudSQL and Cloud Storage

Deploy Kestra to GCP GKE with CloudSQL as a database backend and Google Cloud Storage as internal storage backend.

Overview

This guide provides detailed instructions for deploying Kestra to Google Kubernetes Engine (GKE) with CloudSQL as database backend, and Google Cloud Storage(GCS) for internal storage.

Prerequisites: - Basic command line interface skills. - Familiarity with GCP GKE, PostgreSQL, GCS, and Kubernetes.

Launch an GKE Cluster

First, login to GCP using `gcloud init`.

Run the following command to create an GKE cluster named `my-kestra-cluster`:

```
gcloud container clusters create my-kestra-cluster --region=europe-west3
```

Confirm using the GCP console that the cluster is up.

::alert{type="info"} Before proceeding, check whether the `gke-gcloud-auth-plugin` plugin is already installed:

```
gke-gcloud-auth-plugin --version
```

If the output displays version information, skip this section.

You can install the authentication plugin using:

```
gcloud components install gke-gcloud-auth-plugin
```

::

Run the following command to have your kubecontext point to the newly created cluster:

```
gcloud container clusters get-credentials my-kestra-cluster --region=europe-west3
```

You can now confirm that your kubecontext points to the GKE cluster using:

```
kubectl get svc
```

Install Kestra on GCP GKE

Add the Kestra Helm chart repository and install Kestra:

```
helm repo add kestra https://helm.kestra.io/
helm install my-kestra kestra/kestra
```

Workload Identity Setup

If you are using Google Cloud Workload Identity, you can annotate your Kubernetes service account in the Helm chart configuration. This will allow Kestra to automatically use the associated GCP service account for authentication.

To configure this, you can add the following to your "values.yaml" file:

```
serviceAccount:
  create: true
  name: <your-service-account-name>
  annotations:
    iam.gke.io/gcp-service-account: "<gcp-service-account>@<gcp-project-id>.iam.gserviceacco
```

Alternatively, you can apply the annotation directly when you install Kestra using Helm:

```
helm install my-kestra kestra/kestra \
  --set serviceAccount.annotations.iam.gke.io/gcp-service-account=<gcp-service-account>@<gcp
```

This configuration links your Kubernetes service account to the GCP service account, enabling Workload Identity for secure access to Google Cloud resources.

Launch CloudSQL

1. Go to the Cloud SQL console.
2. Click on `Choose PostgreSQL` (Kestra also supports MySQL, but PostgreSQL is recommended).
3. Put an appropriate Instance ID and password for the admin user `postgres`.
4. Select the latest PostgreSQL version from the dropdown.
5. Choose `Enterprise Plus` or `Enterprise` edition based on your requirements.
6. Choose an appropriate preset among `Production`, `Development` or `Sandbox` as per your requirement.
7. Choose the appropriate region and zonal availability.
8. Hit create and wait for completion.

Enable VM connection to database

db_choices

Figure 1: db_choices

db_setup

Figure 2: db_setup

1. Go to the database overview page and click on `Connections` from the left-side navigation menu.
2. Go to the `Networking` tab, and click on `Add a Network`.
3. In the New Network section, add an appropriate name like `Kestra VM`, and put your GKE pods IP address range in the Network.
4. Click on `Done` in the section.
5. Click on `Save` on the page.

Create database user

1. Go to the database overview page and click on `Users` from the left-side navigation menu.
2. Click on `Add User Account`.
3. Put an appropriate username and password, and click on `Add`.

Create Kestra database

1. Go to the database overview page, and click on `Databases` from the left side navigation menu.
2. Click on `Create Database`.
3. Put an appropriate database name, and click on `Create`.

Update Kestra configuration

Here is how you can configure CloudSQL Database in the Helm chart's values:

```
configuration:
  kestra:
    queue:
      type: postgres
    repository:
      type: postgres
  datasources:
    postgres:
      url: jdbc:postgresql://<your-db-external-endpoint>:5432/<db_name>
      driverClassName: org.postgresql.Driver
```

db_connections

Figure 3: db_connections

db_add_a_network

Figure 4: db_add_a_network

db_users

Figure 5: db_users

```
    username: <your-username>
    password: <your-password>
```

Also, disable the postgres pod by changing `enabled` value in the `postgresql` section from `true` to `false` in the same file.

```
postgresql:
  enabled: false
```

In order for the changes to take effect, run the `helm upgrade` command as:

```
helm upgrade my-kestra kestra/kestra -f values.yaml
```

Prepare a GCS bucket

By default, minio pod is being used as storage backend. This section will guide you on how to change the storage backend to Google Cloud Storage.

By default, internal storage is implemented using the local file system. This section will guide you on how to change the storage backend to Cloud Storage to ensure more reliable, durable, and scalable storage.

1. Go to the Cloud Storage console and create a bucket.
2. Go to IAM and select `Service Accounts` from the left-side navigation menu.
3. On the Service Accounts page, click on `Create Service Account` at the top of the page.
4. Put the appropriate Service account name and Service account description, and grant the service account `Storage Admin` access. Click Done.
5. On the Service Accounts page, click on the newly created service account.
6. On the newly created service account page, go to the `Keys` tab at the top of the page and click on `Add Key.` From the dropdown, select `Create New Key`.
7. Select the Key type as `JSON` and click on `Create`. The JSON key file for the service account will get downloaded.

db_user_creation

Figure 6: db_user_creation

8. We will be using the stringified JSON for our configuration. You can use the bash command `% cat <path_to_json_file> | jq '@json'` to generate stringified JSON.
9. Edit Kestra storage configuration in the Helm chart's values.

Note: If you want to use a Kubernetes service account configured as a workload identify, you don't need to provide anything for `serviceAccount` as it will be autodetected for the pod configuration if it's well configured.

```yaml
configuration:
  kestra:
    storage:
      type: gcs
      gcs:
        bucket: "<your-cloud-storage-bucket-name>"
        projectId: "<your-gcp-project-name>"
        serviceAccount: |
          "<stringified-json-file-contents>"
```

Also, disable the minio pod by changing `enabled` value in the minio section from `true` to `false` in the same file.

```yaml
minio:
  enabled: false
```

In order for the changes to take effect, run the `helm upgrade` command as:

```
helm upgrade my-kestra kestra/kestra -f values.yaml
```

You can validate the storage change from minio to Google Cloud Storage by executing the flow example below with a file and then checking it is uploaded to Google Cloud Storage.

```yaml
id: inputs
namespace: company.team

inputs:
  - id: file
    type: FILE

tasks:
  - id: validator
    type: io.kestra.plugin.core.log.Log
    message: User {{ inputs.file }}
```

Commented-out Examples in values.yaml

To provide users with clear guidance on configuring the values.yaml file, we have included some commented-out examples in the configuration. These examples

can be used to set up various aspects of Kestra, such as secrets, database configurations, and other key parameters. You can uncomment and modify them as needed.

Here's an example of how you can define secrets and other configurations in the values.yaml file:

```yaml
# Example configuration for secrets:
configuration:
  kestra:
    # Configure this section to set secrets for your Kestra instance.
    # secret:
    #   - name: "MY_SECRET_KEY"
    #     value: "my-secret-value"
    #   - name: "ANOTHER_SECRET"
    #     valueFrom:
    #       secretKeyRef:
    #         name: "my-k8s-secret"
    #         key: "my-secret-key"

    # Configure this section to use PostgreSQL as the queue and repository backend.
    # queue:
    #   type: postgres
    # repository:
    #   type: postgres

    # Example of connecting to a PostgreSQL database:
    # datasources:
    #   postgres:
    #     url: jdbc:postgresql://<your-db-endpoint>:5432/<db-name>
    #     driverClassName: org.postgresql.Driver
    #     username: <your-username>
    #     password: <your-password>

# Example to disable default services like MinIO and PostgreSQL if you're using external se
minio:
  # enabled: false
postgresql:
  # enabled: false
```

In this example:

-**Secrets**: You can configure sensitive values as secrets, either hardcoding them or referencing existing Kubernetes secrets.
-**Queue and Repository**: By default, these can use PostgreSQL or any other supported type. Uncomment the relevant lines to use them.
-**PostgreSQL Configuration**: Set the datasources section to provide details for connecting to a PostgreSQL database.

-**Disabling Services**: If you're using external services like CloudSQL or Google Cloud Storage, you can disable the built-in services (MinIO and PostgreSQL).

Feel free to uncomment and modify these examples based on your setup needs. This provides flexibility while keeping your values.yaml well-structured.

Next steps

This guide walked you through installing Kestra to Google GKE with CloudSQL as database and Google Cloud Storage as storage backend.

Reach out via Slack if you encounter any issues or if you have any questions regarding deploying Kestra to production.

Key Value (KV) Store

Build stateful workflows with the KV Store.

Overview

Kestra's workflows are stateless by design. All workflow executions and task runs are isolated from each other by default to avoid any unintended side effects. When you pass data between tasks, you do so explicitly by passing outputs from one task to another and that data is stored transparently in Kestra's internal storage. This stateless execution model ensures that workflows are idempotent and can be executed anywhere in parallel at scale.

However, in certain scenarios, your workflow might need to share data beyond passing outputs from one task to another. For example, you might want to persist data across executions or even across different workflows. This is where the Key Value (KV) store comes into play.

KV Store allows you to store any data in a convenient key-value format. You can create them directly from the UI, via dedicated tasks, Terraform or through the API.

The KV store is a powerful tool that allows you to build stateful workflows and share data across executions and workflows.

How KV Store fits into Kestra's architecture

Kestra's architecture has been designed to offer a transparent separation between the orchestration and data processing capabilities. Kestra's Executor is responsible for executing tasks and workflows without directly interacting with the user's infrastructure. The Executor relies on Workers, which are stateless processes that carry out the computation of runnable tasks and polling triggers. For privacy reasons, workers are the only components that interact with the user's infrastructure, including the internal storage and external services.

Given that data persisted in the KV Store might contain sensitive information, the **KV Store has been built on top of Kestra's internal storage**. This ensures that all values are stored in the your private cloud storage bucket, and

Kestra's database only contains metadata about the object, such as the key, file URI, any attached metadata about the object like TTL, creation date, last updated timestamp, etc.

In short, the KV Store gives you full control and privacy over your data, and Kestra only stores metadata about the KV pairs.

Keys and Values

`Keys` are arbitrary strings. Keys can contain:

- characters in uppercase and or lowercase
- standard ASCII characters

`Values` are stored as ION files in Kestra's internal storage. Values are strongly typed, and can be of one of the following types:

- string
- number
- boolean
- datetime
- date
- duration
- JSON.

For each KV pair, you can set a `Time to Live` (TTL) to avoid cluttering your storage with data that may only be relevant for a limited time.

Namespace binding

Key value pairs are defined at a namespace level and you can access them from the namespace page in the UI in the KV Store tab.

You can create and read KV pairs across namespaces as long as those namespaces are allowed.

UI: How to Create, Read, Update and Delete KV pairs from the UI

Kestra follows a philosophy of Everything as Code and from the UI. Therefore, you can create, read, update, and delete KV pairs both from the UI and Code.

Here is a list of the different ways to manage KV pairs: 1. **Kestra UI**: select a Namespace and go to the KV Store tab — from here, you can create, edit, and delete KV pairs. 2. **Task in a flow**: use the `io.kestra.plugin.core.kv.Set`, `io.kestra.plugin.core.kv.Get`, and `io.kestra.plugin.core.kv.Delete` tasks to create, read, and delete KV pairs in a flow. 3. **Kestra's API**: use our HTTP REST API to create, read, and delete KV pairs. 4. **Kestra's Terraform provider**: use the `kestra_kv` resource to create, read, and delete

edit_delete_kv_pair

Figure 1: edit_delete_kv_pair

KV pairs. 5. **Pebble function**: use the `kv()` function to retrieve a value by key in a flow. 6. **GitHub Actions**: create, read, and delete KV pairs in your CI/CD pipeline.

The sections below provide detailed instructions on how to create and manage KV pairs using each of these methods.

Create new KV pairs from the UI

You can create, read, update, and delete KV pairs from the UI in the following way:

1. Navigate to the `Namespaces` page from the left navigation menu and select the namespace where you want to create the KV pair. navigate_to_namespace
2. Go to the `KV Store` tab. This is where you can see all the KV pairs associated with this namespace. navigate_to_keystore
3. Click on `New Key-Value` button in the top right corner to create a new KV pair. Enter a name for the `Key` and assign a suitable `Type` for the value — it can be a string, number, boolean, datetime, date, duration, or JSON. create_kv_pair
4. Enter the value in the `Value` field.
5. Optionally, you can configure a Time to Live (TTL) for the KV pair. The dropdown contains some standard durations. You can also select `Custom duration` to enter a custom duration as a string in ISO 8601 duration format.
6. Finally, `Save` the changes. Your new KV pair should now be displayed in the list of KV pairs for that namespace.

Update and Delete KV pairs from the UI

You can edit or delete any KV pair by clicking on the `Edit` button on the right side of each KV pair.

CODE: How to Create, Read, Update and Delete KV pairs in your flow code

Create a new KV pair with the `Set` task in a flow

To create a KV pair from a flow, you can use the `io.kestra.plugin.core.kv.Set` task. Here's an example of how to create a KV pair in a flow:

```yaml
id: add_kv_pair
namespace: company.team
```

```yaml
tasks:
  - id: download
    type: io.kestra.plugin.core.http.Download
    uri: https://huggingface.co/datasets/kestra/datasets/raw/main/csv/orders.csv

  - id: set_kv
    type: io.kestra.plugin.core.kv.Set
    key: my_key
    value: "{{ outputs.download.uri }}"
    namespace: company.team # the current namespace of the flow is used by default
    overwrite: true # whether to overwrite or fail if a value for that key already exists;
    ttl: P30D # optional Time to Live (TTL) for the KV pair

  - id: set_simple_kv
    type: io.kestra.plugin.core.kv.Set
    key: simple_string
    value: hello from Kestra

  - id: set_json_kv
    type: io.kestra.plugin.core.kv.Set
    key: json_kv
    value: |
      {
        "author": "Rick Astley",
        "song": "Never Gonna Give You Up"
      }

  - id: get_kv
    type: io.kestra.plugin.core.output.OutputValues
    values:
      my_key: "{{ kv('my_key') }}"
      simple_string: "{{ kv('simple_string') }}"
      favorite_song: "{{ json(kv('json_kv')).song }}"
```

You can use the `io.kestra.plugin.core.kv.Set` task to create or modify any KV pair. When modifying existing values, you can leverage the `overwrite` boolean parameter to control whether to overwrite the existing value or fail if a value for that key already exists. By default, the `overwrite` parameter is set to `true` so that the existing value is always updated.

Read KV pairs with Pebble

The easiest way to retrieve a value by key is to use the `{{ kv('YOUR_KEY') }}` Pebble function.

Here is the full syntax of that function:

`{{ kv(key='your_key_name', namespace='your_namespace_name', errorOnMissing=false) }}`

Assuming that you retrieve the key in a flow in the same namespace as the one for which the key was created, you can simply use `"{{ kv('my_key') }}"` to retrieve the value:

```yaml
id: read_kv_pair
namespace: company.team
tasks:
  - id: log_key
    type: io.kestra.plugin.core.log.Log
    message: "{{ kv('my_key') }}"
```

When retrieving the key from another namespace, you can use the following syntax:

```yaml
id: read_kv_pair_from_another_namespace
namespace: company.team
tasks:
  - id: log_key_from_another_namespace
    type: io.kestra.plugin.core.log.Log
    message: "{{ kv('my_key', 'kestra.engineering.myproject') }}"
```

By default, when you try to retrieve a key that doesn't exist, the task using the `"{{ kv('non_existing_key') }}"` expression will run without errors — that expression will simply return `null`. If you prefer to instead throw an error when the key doesn't exist, you can set the `errorOnMissing` parameter to `true`:

```yaml
id: read_non_existing_kv_pair
namespace: company.team
tasks:
  - id: log_key_from_another_namespace
    type: io.kestra.plugin.core.debug.Return
    format: "{{ kv('non_existing_key', errorOnMissing=true) }}"
```

The function arguments such as the `errorOnMissing` keyword can be skipped for brevity as long as you fill in all positional arguments i.e. `{{ kv(key='your_key_name', namespace='your_namespace_name', errorOnMissing=false) }}` — the version below will have the same effect: {{ kv(key='my_key', namespace='company.team') }}

```yaml
id: read_non_existing_kv_pair
namespace: company.team
tasks:
  - id: log_key_from_another_namespace
    type: io.kestra.plugin.core.debug.Return
    format: "{{ kv('my_key', 'kestra.engineering.myproject', true) }}"
```

Read KV pairs with the `Get` task

You can also retrieve the value of any KV pair using the `Get` task. The `Get` task will produce the `value` output, which you can use in subsequent tasks. This option is a little more verbose but it has two benefits: 1. More declarative syntax. 2. Useful when you need to pass the current state of that value to multiple downstream tasks.

```yaml
id: get_kv_pair
namespace: company.team

tasks:
  - id: get
    type: io.kestra.plugin.core.kv.Get
    key: my_key
    namespace: company.team
    errorOnMissing: false

  - id: log_key_get
    type: io.kestra.plugin.core.log.Log
    message: "{{ outputs.get.value }}"
```

Read and parse JSON-type values from KV pairs

To parse JSON values in Kestra's templated expressions, make sure to wrap the `kv()` call in the `json()` function, e.g. `"{{ json(kv('your_json_key')).json_property }}"`.

The following example demonstrates how to parse values from JSON-type KV pairs in a flow:

```yaml
id: kv_json_flow
namespace: company.team

tasks:
  - id: set_json_kv
    type: io.kestra.plugin.core.kv.Set
    key: favorite_song
    value: |
      {
        "author": "Rick Astley",
        "song": "Never Gonna Give You Up",
        "album": {
          "name": "Whenever You Need Somebody",
          "release_date": "1987-11-16"
        }
      }
```

```yaml
  - id: parse_json_kv
    type: io.kestra.plugin.core.log.Log
    message:
      - "Author: {{ json(kv('favorite_song')).author }}"
      - "Song: {{ json(kv('favorite_song')).song }}"
      - "Album name: {{ json(kv('favorite_song')).album.name }}"
      - "Album release date: {{ json(kv('favorite_song')).album.release_date }}"

  - id: get
    type: io.kestra.plugin.core.kv.Get
    key: favorite_song

  - id: parse_json_from_kv
    type: io.kestra.plugin.core.log.Log
    message: "Country: {{ json(outputs.get.value).album.name }}"
```

Read keys by prefix with the `GetKeys` task

If you want to check if some values already exist for a given key, you can search keys by prefix:

```yaml
id: get_keys_by_prefix
namespace: company.team

tasks:
  - id: get
    type: io.kestra.plugin.core.kv.GetKeys
    prefix: "test_"
    namespace: company.team

  - id: log_key_prefix
    type: io.kestra.plugin.core.log.Log
    message: "{{ outputs.get.keys }}"
```

The output will be a list of keys - if no keys were found, an empty list will be returned.

Delete a KV pair with the `Delete` task

The `io.kestra.plugin.core.kv.Delete` task will produce the boolean output `deleted` to confirm whether a given KV pair was deleted or not.

```yaml
id: delete_kv_pair
namespace: company.team

tasks:
  - id: kv
    type: io.kestra.plugin.core.kv.Delete
```

```
    key: my_key
    namespace: company.team
    errorOnMissing: false

  - id: check_if_deleted
    type: io.kestra.plugin.core.log.Log
    message: "{{ outputs.kv.deleted }}"
```

API: How to Create, Read, Update and Delete KV pairs via REST API

Let's look at how you can interact with the KV Store via the REST API.

Create a KV pair

The API call to set the KV pair follows the structure:

```
curl -X PUT -H "Content-Type: application/json" http://localhost:8080/api/v1/namespaces/{nam
```

For example:

```
curl -X PUT -H "Content-Type: application/json" http://localhost:8080/api/v1/namespaces/comp
```

The above `curl` command will create the KV pair with key `my_key` and the `Hello World` string value in the `company.team` namespace. The API does not return any response.

Read the value by key

You can get any particular KV pair using:

```
curl -X GET -H "Content-Type: application/json" http://localhost:8080/api/v1/namespaces/{nam
```

For example:

```
curl -X GET -H "Content-Type: application/json" http://localhost:8080/api/v1/namespaces/comp
```

This will retrieve a KV pair with the key `my_key` in the `company.team` namespace. The output of the API will contain the data type of the value and the retrieved value of the KV pair:

```
{"type": "STRING", "value": "Hello World"}
```

Read all keys in the namespace

You can list all keys in the namespace as follows:

```
curl -X GET -H "Content-Type: application/json" http://localhost:8080/api/v1/namespaces/{nam
```

The `curl` command below will return all keys in the `company.team` namespace:

```
curl -X GET -H "Content-Type: application/json" http://localhost:8080/api/v1/namespaces/comp
```

The output will be returned as a JSON array of all keys in the namespace:

```
[
  {"key":"my_key","creationDate":"2024-07-27T06:10:33.422Z","updateDate":"2024-07-27T06:11:0
  {"key":"test_key","creationDate":"2024-07-27T04:37:18.196Z","updateDate":"2024-07-27T04:37
]
```

Delete a KV pair

You can delete any KV pair using the following API call:

```
curl -X DELETE -H "Content-Type: application/json" http://localhost:8080/api/v1/namespaces/t
```

This call returns a boolean indicating whether the key was deleted.

For example, the following `curl` command will return `false` because the key `non_existing_key` does not exist:

```
curl -X DELETE -H "Content-Type: application/json" http://localhost:8080/api/v1/namespaces/c
```

However, when we try to delete a key `my_key` which exists in the `company.team` namespace, the same API call will return `true`:

```
curl -X DELETE -H "Content-Type: application/json" http://localhost:8080/api/v1/namespaces/c
```

TERRAFORM: How to Create, Read, Update and Delete KV pairs via Terraform

Create a KV pair

You can create a KV pair via Terraform by using the `kestra_kv` resource.

Here is an example of how to create a KV pair:

```
resource "kestra_kv" "my_key" {
  namespace = "company.team"
  key       = "my_key"
  value     = "Hello Woprld"
  type      = "STRING"
}
```

Read a KV pair

You can read a KV pair via Terraform by using the `kestra_kv` data source.

Here is an example of how to read a KV pair:

```
data "kestra_kv" "new" {
  namespace = "company.team"
```

```
  key       = "my_key"
}
```

As with anything in Terraform, you can manage the state of your KV resources by adjusting the Terraform code and running the `terraform apply` command to create, update, or delete your KV pairs.