nstack Documentation

Release 0.1.x

Mandeep Gill & Leo Anthias

Apr 13, 2017

Contents

Welcome to the NStack Documentation! Please also look at our main GitHub page , and our online examples.

CHAPTER 1

What is NStack?

- Productionise your models and data integration code to the cloud as reusable functions that can be monitored, versioned, shared, and updated
- Effortlessly compose workflows to integrate your functions with your data and event sources, without dealing with any infrastructure
- NStack containerises and orchestrates your workflows on your cloud provider and makes sure they always work

CHAPTER 2

Contents

NStack Uses

Productionising Models

Productionise your models in the cloud without complex engineering, where they can be used in workflows and attached to data-sources. For instance, you can build a Random Forest classifier locally in Python, publish it to your cloud provider, and connect it to a streaming system, database, data-lake, or HTTP endpoint in under 10 minutes.

Data Integration

Transform disparate and disconnected data-sources – such as 3rd-party APIs, legacy infrastructure, or databases into streams of typed, structured records – which can be composed together. For instance, you could set up a workflow in the cloud which pipes the Twitter Ads API into your data-lake (and even do some modelling in Python in-transit) in under 5 minutes.

Software Lifecycle

NStack provides best practices from software engineering and end-to-end software life-cycle management to the data science process, including,

- **sharing and reuse** - build and deploy individual model modules to your cloud that can reused, imported, and utilised by other members, either within your team or by third-parties
- **reproducibility** - build models with guaranteed versions of system and language dependencies, that can then be shared with confidence that your code is bit-for-bit identical everywhere
- **versioning** - modules a have a immutable, globally unique, version - so when importing to use in a workflow, you can be sure the API, code, and artifacts are just as you intended, allowing you to upgrade to newer versions when ready

- **runtime isolation** - all modules in your workflow run using our container technology, meaning they are fully isolated from each other, can scale up as needed, and won't interfere with each others data

NStack Concepts

Example

We can express this within the NStack scripting language locally as follows (it can help to think of it akin to Bash-style piping for microservices),

```
module Demo:0.1.0 {
  import NStack.Transformers:0.1.4 as T
  import Acme.Classifiers:0.3.0 as C

  // our analytics workflow
  def workflow = Sources.Postgresql<(Text, Int)>
                 | T.transform { strength = 5 }
                 | C.classify { model = "RandomForest" }
                 | Sinks.S3blob<Text>
}
```

Intro Screencast

Modules

A *module* is a piece of code that has been deployed to NStack, either by you or someone else. It has an input schema and an output schema, which defines what kind of data it can receive, and the kind of data that it returns.

Sources & Sinks

- A *source* is something which emits a stream of data.
- A *sink* is something which can receive a stream of data.

Example sources and sinks are databases, files, message-queues, and HTTP endpoints. Like modules, you can define the input and output schemas for your sources and sinks.

Workflows

Modules, sources, and sinks can be combined together to build *workflows*. This is accomplished using the *NStack Workflow Language*, a simple, high-level language for connecting things together on the *NStack Platform*.

Processes

When a workflow is started, it becomes a running *process*. You can have multiple processes of the same workflow.

Installation & Upgrading

Installation

NStack is platform-agnostic and can run out-of-the-box wherever you can run a virtual machine, including:

- your cloud provider of choice
- your internal cloud
- locally using VirtualBox, VMWare, or your operating system's native virtualisation

For Proof of Concepts, NStack offers a hosted solution. If this is required, please reach out to info@nstack.com.

The virtual appliance can be found on the NStack's GitHub Releases page, where it is provided as a `raw` image. We also provide an AWS AMI, which can be found under the id of `ami-53a47245`. Basic install instructions can be found on our [GitHub page](www.github.com/nstack/nstack).

To launch this AMI to an EC2 instance on your AWS account, you can click here.

The *NStack Server* is configured using `cloud-init`, which is supported by major cloud providers, so it will work out of the box with your credentials. Please note that the first time you boot NStack it may take a few minutes to initialise.

Upgrading

The *NStack Server* can be updated atomically using *rpm-ostree*. To upgrade to a new release, you can simply run:

```
> sudo rpm-ostree upgrade
```

Following the upgrade, you should reboot your machine with:

```
> sudo reboot
```

NStack releases follow a monthly cadence.

Quick Tutorial

In this section, we're going to see how to build up a simple NStack module, deploy it to the cloud, and use it in a workflow by connecting it to a *source* and a *sink*.

Intro Screencast

The following screencast accompanies this tutorial, demonstrating building a module and connecting it within a workflow,

By the end of the tutorial, you will learn how to publish your code to NStack and connect it to event and data sources. Enjoy!

Note: To learn more about modules, sources, and sinks, read *Concepts*

Make sure you have *installed NStack* and let's get going. These instructions are for the Linux and macOS versions of the NStack CLI, so adjust accordingly if you are on Windows.

Building a Module

Modules contain the *functions* that you want to publish to the NStack platform.

After this tutorial, we will have a simple Python module deployed to our NStack instance. This module will have a single function in it which counts the number of characters in some text.

Note: Before starting, check that NStack is installed by running `nstack --version` in your terminal. If you got information about the version of NStack you have, you're good to go. If that didn't work, check out *Installation* again.

Step 1: `init`

We want to create a new Python module.

Create a directory called `Demo` where you would like to build your module and `cd` into that directory using your terminal. NStack uses the name of the directory as the default name of the module

To create a new module, run `nstack init python`. You should see the following output confirming that this operation was successful.

```
~> mkdir Demo
~> cd Demo
~/Demo> nstack init python
python module 'Demo:0.0.1-SNAPSHOT' successfully initialised at ~/Demo
```

Because NStack versions your modules, it has given `Demo` a version number (`0.0.1-SNAPSHOT`). Because the version number has a `SNAPSHOT` appended to it, this means NStack allows you to override it every time you build. This is helpful for development, as you do not need to constantly increase the version number. When you deem your module is ready for release, you can remove `SNAPSHOT` and NStack will create an immutable version of `0.0.1`.

A successful `init` will have created some files.

```
~/Demo> ls
nstack.yaml  requirements.txt  service.py  setup.py
```

This is the skeleton of an NStack module. `nstack.yaml` is the configuration file for your module, and `service.py` is where the code of your module lives (in this case, it's a Python class). `requirements.txt` and `setup.py` are both standard files for configuring Python.

We're going to be concerned with `nstack.yaml` and `service.py`. For a more in-depth look at all these files, refer to *Module Structure*.

In `service.py`, there is a `Service` class. This is where we write the functions we want to use on NStack. It is pre-populated with a sample function, `numChars`, that counts the number of characters in some text.

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Demo Service
"""
import nstack

class Service(nstack.BaseService):
    def numChars(self, x):
        return len(x)
```

`nstack.yaml` is where the configuration for this module lives. NStack fills in the `service`, `stack`, and `parent` for you, so we don't need to worry about them for now.

```
# Service name (a combination of lower case letters, numbers, and dashes)
name: Demo:0.0.1-SNAPSHOT

# The language stack to use
stack: python

# Parent Image
parent: NStack.Python:0.25.0

api: |
  numChars : Text -> Integer
```

We're going to focus on the `api` section, where you tell NStack which of the functions in `service.py` you want to publish as functions on NStack, and their input and output schemas (also known as types). In this instance, we are telling NStack to publish one function, `numChars`, which takes `Text` and returns an `Integer`.

Note: The schema – or type – system is a key feature of NStack that lets you define the sort of data your function can take as input, and produce as output. This helps you ensure that your module can be reused and works as intended in production.

Step 2: `build`

To build and publish our module on NStack, we use the `build` command.

```
~/Demo> nstack build
Building NStack Container module Demo:0.0.1-SNAPSHOT. Please wait. This may take some␣
↪time.
Module Demo:0.0.1-SNAPSHOT built successfully. Use `nstack list functions` to see all␣
↪available functions
```

When we run `build`, the code is packaged up and sent to the server.

We can check that our `numChars` function is live by running the suggested `nstack list functions` command:

```
~/Demo> nstack list functions
Demo:0.0.1-SNAPSHOT
  numChars : Text -> Integer
```

That's it! Our `numChars` function is live in the cloud, and is ready to be connected to input and output data streams, which the next tutorial will cover.

Building a Workflow

In the previous tutorial, we built and published a Python module, *Demo*. This module had a single function on it, `numChars`, which counted the number of characters in some text. Although it has been published, it needs to be connected to a *source* and a *sink*.

Note: Sources generate data which gets sent to your function, and sinks receive the data which your function outputs. Learn more in *Concepts*

Let's refresh ourselves on what the input and output types of our function were by asking NStack:

```
> nstack list functions
Demo:0.0.1-SNAPSHOT.numChars : Text -> Integer
```

This means that our function can be connected to any source which generates `Text`, and can write to any sink which can take an `Integer` as input.

One of the sources that NStack provides is `http`; if you use this source, NStack sets up an HTTP endpoint which you can send `JSON`-encoded data to. As a sink, we are going to use the NStack `log`, which is a sink for seeing the output from your function. We are going to use these two integrations in our tutorial.

Note: See a list of available sources and sinks in Supported Integrations

Creating a workflow module

To write workflows, we create a special NStack workflow module, which we create in the same way we create a Python module – by using `init`.

Let's create a new directory called `DemoWorkflow`, `cd` into the directory, and create a new workflow module.

```
~/DemoWorkflow/ nstack init workflow
Module 'DemoWorkflow:0.0.1-SNAPSHOT' successfully initialised at /home/nstack/Demo/
↪DemoWorkflow
```

`init` has created a single file, `workflow.nml`, which is where we write our workflow module using NStack's scripting language. If we look inside the file, we see that NStack has created an example module for us.

Note: Just like Python modules, workflow modules are versioned.

```
module DemoWorkflow:0.0.1-SNAPSHOT {
  import Python.Hello:0.0.1 as Hello;
  // A sample workflow
  def w = Sources.http<Text> { http_path = "/demo" } | Hello.numChars | Sinks.log
↪<Integer>;
}
```

This currently has a single workflow on it, `w`, which uses a function imported from a module called `Python.Hello` with the version number of `0.0.1`. Like the workflow we will create, this example workflow creates an HTTP endpoint which pipes data to a function, and pipes data from the function to the NStack log.

When we created our Python module, we defined the input and output types of our function in our API. On NStack, sources and sinks also have types: this workflow specifies that the HTTP source only receives and passes on `Text`, and the log only accepts `Integers`. Because our Python function takes `Text`, counts it, and returns `Integers`, that means it can fit in the middle of the workflow.

Writing our workflow

First, let's change the import statement to import our *Demo* module instead of *Python.Hello*.

```
import Demo:0.0.1-SNAPSHOT as Demo;
```

Now, let's change our workflow to use the `numChars` function on `Demo`, instead of the one on `Python.Hello`.

```
def w = Sources.http<Text> { http_path = "/demo" } | Demo.numChars | Sinks.log
↪<Integer>;
```

Note: The http source is configured in this example to expose an endpoint on `/demo`. If you are using the demo server, we would recommend changing `/demo` to something more unique – as someone else may have already taken that endpoint.

Let's break these parts to see what we're doing:

Part	Description
`Sources.http<Text> { http_path = "/demo" }`	Use `http` as a source, which creates an endpoint on `/demo`. The `<Text>` statement means it can only accept and pass on Text.
`Demo.numChars`	The name of the function which we built.
`Sinks.log<Integer>`	Use NStack's log as a sink. The `<Integer>` statement means it can only accept Integers.

NStack uses the `|` operator to connect statements together, just like in a shell such as `bash`.

Building our workflow

Before we start our workflow, we need to build it in the cloud with NStack. We do this in the same way we build a Python module. We save our `workflow.nml` file and run:

```
~/DemoWorkflow/ nstack build
Building NStack Workflow module DemoWorkflow:0.0.1-SNAPSHOT. Please wait. This may␣
↪take some time.
Workflow module DemoWorkflow:0.0.1-SNAPSHOT built successfully. Use `nstack list all`␣
↪to see all available functions.
```

We can now see our workflow is live by using the list command.

```
~/DemoWorkflow/ nstack list workflows
DemoWorkflow:0.0.1-SNAPSHOT
  w : Workflow
```

This means our workflow is ready to be started.

Starting and using our workflow

To start our workflow in the cloud, we use the start command:

```
~/DemoWorkflow/ $ nstack start DemoWorkflow:0.0.1-SNAPSHOT.w
```

We now have a live HTTP endpoint on `localhost:8080/demo`. The HTTP endpoint is configured to accept JSON-encoded values. We defined it to use an input schema of `Text`, so we will be able to send it any JSON `string`. In our JSON, we put `params` as the key, and our input as the value:

We can call it using `nstack send`:

```
~/DemoWorkflow/ $ nstack send "/demo" '"Foo"'
> Message Accepted
```

When workflows are started, they become *processes* which have numerical identifiers (_ids_). We can see the id of our process by running:

```
~/DemoWorkflow/ $ nstack ps
1
```

And if we look at the log of our process, which we configured as the sink, we will be able to see the result. Because our process was started with an id of `1`, we run the following:

```
> nstack log 1
Feb 17 09:59:26 nostromo nstack-server[8925]: OUTPUT: 3
```

Great - we can see that the output of our function (and the number of characters in "Foo") is 3.

In-Depth Tutorial - Productionising a Classifier

In this section, we're going to productionise a Random Forest classifier written with sklearn, deploy it to the cloud, and use it in a more sophisticated workflow.

By the end of the tutorial, you will learn how to build modules with dependencies, write more sophisticated workflows, and build abstractions over data-sources. Enjoy!

So far, we have built and published a Python module with a single function on it, `numChars`, and built a workflow which connects our function to an HTTP endpoint. This in itself isn't particularly useful, so, now that you've got the gist of how NStack works, it's time to build something more realistic!

In this tutorial, we're going to create and productionise a simple classifier which uses the famous iris dataset. We're going to train our classifier to classify which species an iris is, given measurements of its sepals and petals. You can find the dataset we're using to train our model here.

First, let's look at the the format of our data to see how we should approach the problem. We see that we have five fields:

Field Name	Description	Type
`species`	The species of iris	Text
`sepal_width`	The width of the sepal	Double
`sepal_length`	The length of the sepal	Double
`petal_width`	The width of the petal	Double
`petal_length`	The length of the petal	Double

If we are trying to find the species based on the sepal and petal measurements, this means these measurements are going to be the input to our classifier module, with text being the output. This means we need to write a function in Python which takes four `Doubles` and returns `Text`.

Creating your classifier module

To begin, let's make a new directory called `Iris.Classify`, `cd` into it, and initialise a new Python module:

```
~/ $ mkdir Iris.Classify; cd Iris.Classify
~/Iris.Classify/ $ nstack init python
python module 'Iris.Classify' successfully initialised at ~/Iris.Classify
```

Next, let's download our training data into this directory so we can use it in our module. We have hosted it for you as a CSV on GitHub.

```
~/Iris.Classify/ $ curl -O https://raw.githubusercontent.com/nstackcom/nstack-
↪examples/master/iris/Iris.Classify/train.csv
```

Defining our API

As we know what the input and output of our classifier is going to look like, let's edit the `api` section of `nstack.yaml` to define our API (i.e. the entry-point into our module). By default, a new module contains a sample function `numChars`, which we replace with our definition. We're going to call the function we write in Python `predict`, which means we fill in the `api` section of `nstack.yaml` as follows:

```
api : |
    predict : (Double, Double, Double, Double) -> Text
```

This means we want to productionise a single function, `predict`, which takes four `Doubles` (the measurements) and returns `Text` (the iris species).

Writing our classifier

Now that we've defined our API, let's jump into our Python module, which lives in `service.py`. We see that NStack has created a class `Service`. This is where we add the functions for our module. Right now it also has a sample function in it, `numChars`, which we can remove.

Let's import the libaries we're using.

```
import nstack
import pandas as pd

from sklearn.ensemble import RandomForestClassifier
```

Note: Python modules must also import `nstack`

Before we add our `predict` function, we're going to add `__init__`, the Python constructor function which runs upon the creation of our module. It's going to load our data from `train.csv`, and use it to train our Random Forest classifier:

```
def __init__(self):
    train = pd.read_csv("train.csv")

    self.cols = ['petal_length', 'petal_width', 'sepal_length', 'sepal_width']
    colsRes = ['class']
    trainArr = train.as_matrix(self.cols)
    trainRes = train.as_matrix(colsRes)

    rf = RandomForestClassifier(n_estimators=100)
    rf.fit(trainArr, trainRes)
    self.rf = rf
```

Now we can write our `predict` function. The second argument, `inputArr`, is the input – in this case, our four `Doubles`. To return text, we simply return from the function in Python.

```
def predict(self, inputArr):
    points = [inputArr]
    df = pd.DataFrame(points, columns=self.cols)
    results = self.rf.predict(df)
    return results.item()
```

Configuring your module

When your module is started, it is run in a Linux container on the NStack server. Because our module uses libraries like `pandas` and `sklearn`, we have to tell NStack to install some extra operating system libraries inside your module's container. NStack lets us specify these in our `nstack.yaml` configuration file in the `packages` section. Let's add the following packages:

```
packages: ['numpy', 'python3-scikit-learn', 'scipy', 'python3-scikit-image', 'python3-
↪pandas']
```

Additionally, we want to tell NStack to copy our `train.csv` file into our module, so we can use it in `__init__`. `nstack.yaml` also has a section for specifying files you'd like to include:

```
files: ['train.csv']
```

Publishing and starting

Now we're ready to build and publish our classifier. Remember, even though we run this command locally, our module gets built and published on your NStack server in the cloud.

```
~/Iris.Classify/ $ nstack build
Building NStack Container module Iris.Classify. Please wait. This may take some time.
Module Iris.Classify built successfully. Use `nstack list functions` to see all␣
↪available functions.
```

We can now see `Iris.Classify.predict` in the list of existing functions (along with previously built functions like `demo.numChars`),

```
~/Iris.Classify/ $ nstack list functions
 Iris.Classify:0.0.1-SNAPSHOT
     predict : (Double, Double, Double, Double) -> Text
 Demo:0.0.1-SNAPSHOT
     numChars : Text -> Integer
```

Our classifier is now published, but to use it we need to connect it to an event source and sink. In the previous tutorial, we used HTTP as a source, and the NStack log as a sink. We can do the same here. This time, instead of creating a workflow module right away, we can use nstack's `notebook` command to test our workflow first. `notebook` opens an interactive shell where we can write our workflow. When we are finished, we can `Ctrl-D`.

```
~/Iris.Classify/ $ nstack notebook
import Iris.Classify:0.0.1-SNAPSHOT as Classifier;
Sources.http<(Double, Double, Double, Double)> | Classifier.predict | Sinks.log<Text>
[Ctrl-D]
```

This creates an HTTP endpoint on `http://localhost:8080/irisendpoint` which can receive four `Doubles`, and writes the results to the log as `Text`. Let's check it is running as a process:

```
~/Iris.Classify/ $ nstack ps
1
2
```

In this instance, it is running as process 2. We can test our classifier by sending it some of the sample data from train.csv.

```
~/Iris.Classify/ $ nstack send "/irisendpoint" '[4.7, 1.4, 6.1, 2.9]'
Message Accepted
~/Iris.Classify/ $ nstack log 2
Feb 17 10:32:30 nostromo nstack-server[8925]: OUTPUT: "Iris-versicolor"
```

Our classifier is now productionised.

Features

In this section, we're going to describe some of the more advanced features you can do with NStack when building your modules and composing them together to build workflows.

Composition

Workflows can contain as many steps as you like, as long as the output type of one matches the input type of the other. For instance, let's say we wanted to create the following workflow based on the Iris example in *In-Depth Tutorial - Productionising a Classifier* and available on GitHub

- Expose an HTTP endpoint which takes four `Doubles`
- Send these `Doubles` to our classifier, `Iris.Classify`, which will tell us the species of the iris
- Count the number of characters in the species of the iris using our `Demo.numChars` function
- Write the result to the log

We could write the following workflow:

```
module Iris.Workflow:0.0.1-SNAPSHOT {
  import Iris.Classify:0.0.1-SNAPSHOT as Classifier;
  import Demo:0.0.1-SNAPSHOT as Demo;

  def multipleSteps = Sources.http<(Double, Double, Double, Double)> { http_path = "/
↪irisendpoint" } | Classifier.predict | Demo.numChars | sinks.log<Integer>;
}
```

Note: `numChars` and `predict` can be *composed* together because their types – or schemas – match. If `predict` wasn't configured to output `Text`, or `numChars` wasn't configured to take `Text` as input, NStack would not let you build the following workflow.

Workflow Reuse

All of the workflows that we have written so far have been *fully composed*, which means that they contain a source and a sink. Many times, you want to split up sources, sinks, and functions into separate pieces you can share and reuse. In this case, we say that a workflow is *partially composed*, which just means it does not contain a source and a sink. These workflows cannot be `start`ed by themselves, but can be shared and attached to other sources and/or sinks to become *fully composed*.

For instance, we could combine `Iris.Classify.predict` and `demo.numChars` from the previous example to form a new workflow `speciesLength` like so:

```
module Iris.Workflow:0.0.1-SNAPSHOT {
  import Iris.Classify:0.0.1-SNAPSHOT as Classifier;
  import Demo:0.0.1-SNAPSHOT as Demo;

  def speciesLength = Classifier.predict | Demo.numChars;
}
```

Because our workflow `Iris.Workflow.speciesLength` has not been connected to a source or a sink, it in itself is still a function. If we build this workflow, we can see `speciesLength` alongside our other functions by using the `list` command:

```
~/Iris.Workflow/ $ nstack list functions
Iris.Classify:0.0.1-SNAPSHOT
  predict : (Double, Double, Double, Double) -> Text
Demo:0.0.1
  numChars : Text -> Integer
Iris.Workflow:0.0.1-SNAPSHOT
  speciesLength : (Double, Double, Double, Double) -> Integer
```

As we would expect, the input type of the workflow is the input type of `Iris.Classify.predict`, and the output type is the output type of `demo.numChars`. Like other functions, this must be connected to a source and a sink to make it *fully composed*, which means we could use this workflow it in *another* workflow.

```
module Iris.Endpoint:0.0.1-SNAPSHOT {
  import Iris.Workflow:0.0.1-SNAPSHOT as IrisWF;
  def http = Sources.http<(Double, Double, Double, Double)> | IrisWF.speciesLength |␣
↪Sinks.log<Integer>;
}
```

Often times you want to re-use a source or a sink without reconfiguring them. To do this, we can similarly separate the sources and sinks into separate workflows, like so:

```
module Iris.Workflow:0.0.1-SNAPSHOT {
  import Iris.Classify:0.0.1-SNAPSHOT as Classifier;

  def httpEndpoint = sources.http<(Double, Double, Double, Double)> { http_path =
↪"speciesLength" };
  def logSink = sinks.log<Text>;

  def speciesWf = httpEndpoint | Classifier.predict | logSink;
}
```

Separating sources and sinks becomes useful when you're connecting to more complex integrations which you don't want to configure each time you use it – many times you want to reuse a source or sink in multiple workflows. In the following example, we are defining a module which provides a source and a sink which both sit ontop of Postgres.

```
module Iris.DB:0.0.1-SNAPSHOT {
  def petalsAndSepals = Sources.postgres<(Double, Double, Double, Double)> {
    pg_database = "flowers",
    pg_query = "SELECT * FROM iris"
  };

  def irisSpecies = Sinks.postgres<Text> {
    pg_database = "flowers",
    pg_table = "iris"
  };
}
```

If we built this module, `petalsAndSepals` and `irisSpecies` could be used in other modules as sources and sinks, themselves.

We may also want to add a functions to do some pre- or post- processing to a source or sink. For instance:

```
module IrisCleanDbs:0.0.1-SNAPSHOT {

  import PetalTools:1.0.0 as PetalTools;
  import TextTools:1.1.2 as TextTools;
  import Iris.DB:0.0.1-SNAPSHOT as DB;

  def roundedPetalsSource = DB.petalsAndSepals | PetalsTools.roundPetalLengths;
  def irisSpeciesUppercase = TextTools.toUppercase | DB.irisSpecies;
}
```

Because `roundedPetalsSource` is a combination of a source and a function, it is still a valid source. Similarly, `irisSpeciesUppercase` is a combination of a function and a sink, so it is still a valid sink.

Because NStack functions, source, and sinks can be composed and reused, this lets you build powerful abstractions over infrastructure.

Versioning

Modules in NStack are versioned with a 3-digit suffix that is intended to follow semantic versioning, e.g.:

```
Demo:0.0.1
```

This is specified in the `nstack.yaml` for code-based modules, and in `workflow.nml` for workflow modules. A module of a specific version is completely immutable, and it's not possible to build another copy of the module with the same version without deleting it first.

Snapshots

When creating a new module, i.e. with `nstack init`, your module will have the version number (`0.0.1-SNAPSHOT`). The `SNAPSHOT` tag tells NStack to allow you to override it every time you build. This is helpful for development, as you do not need to constantly increase the version number. When you deem your module is ready for release, you can remove `SNAPSHOT` and NStack will create an immutable version of `0.0.1`.

Configuration

In addition to receiving input at runtime, modules, sources, and sinks often need to be able to configured by a workflow author. To do this, we use brackets and pass in a list of named records:

```
Sources.Postgres<Text> {
     pg_host = "localhost",
     pg_port = "5432",
     pg_user = "user",
     pg_password = "123456",
     pg_database = "db",
     pg_query = "SELECT * FROM tbl;"
 }
```

For sources and sinks, some parameters are mandatory, and some provide sensible defaults. This is documented in Supported Integrations.

To pass configuration parameters to a module, we use the same syntax

```
FirstLastName.full_name { first_name = "John" }
```

NStack passes in configuration parameters as a dictionary, `args`, which is added to the base class of your module. For instance, in Python you can access configuration parameters in the following manner:

```
class Service(nstack.BaseService):

    def full_name(self, second_name):
      full_name = "{} {}".format(self.args.get("first_name", "Tux"), second_name)
      return full_name
```

Framework Modules

It is often useful to create a common parent module with dependencies already installed, either to save time or for standardisation. NStack supports this with *Framework Modules*. Simply create a new module similar to above, `nstack init framework [parent]`, and modify the resulting `nstack.yaml` as needed.

You can then build this module using `nstack build`, and refer to it from later modules within the `parent` field of their `nstack.yaml` config file.

Supported Languages

NStack is language-agnostic and allows you to write modules in multiple languages and connect them together – currently we support Python, with R and Java coming soon. More languages will be added over time – if there's something you really need, please let us know!

Python

Basic Structure

NStack services in Python inherit from a base class, called `BaseService` within the `nstack` module:

```
import nstack

class Service(nstack.BaseService):
    def numChars(self, msg):
        return len(msg)
```

Note: Ensure you import the nstack module in your service, e.g. `import nstack`

Any function that you export within the `API` section of your `nstack yaml` must exist as a method on this class (you can add private methods on this class for internal use as expected in Python).

Data comes into this function via the method arguments - for `nstack` all the data is passed within a single argument that follows the `self` parameter. For instance, in the example above there is a single argument `msg` consisting of a single `string` element that may be used as-is. However if your function was defined as taking `(Double, Double)`, you would need to unpack the tuple in Python first as follows,

```python
def test(self, msg):
    x, y = msg
    ...
```

Similarly, to return data from your NStack function, simply return it as a single element within your Python method, as in the top example above.

The NStack object lasts for the life-time of the workflow, so if there is any initialisation you need to do within your service you can perform this within the object `__init__` method, e.g. open a database connection, load a data-set. However remember to call the parent object `__init__` method to ensure NStack is initialised correctly, i.e..

```python
import nstack

class Service(nstack.BaseService):
    def __init__(self):
        super().__init__()
        # custom initialisation here
```

Notes

- Anything your``print`` will show up in `nstack log` to aid debugging. (all output on `stdout` and `stderr` is sent to the NStack logs)
- Extra libraries from pypi using `pip` can be installed by adding them to the `requirements.txt` file in the project directory - they will be installed during `nstack build`

R

Coming soon

Java

Coming soon

Reference

nstack CLI

The nstack CLI is used to build modules and workflows on a linked NStack server. It can be configured via the `nstack.conf yaml` file found in `~\.config` on Linux/macOS, and in `HOME/AppUser/Roaming` on Windows, or indirectly via the `nstack set-server` command described below.

Usage

Having installed the CLI, make sure it's accessible from your path

```
$ nstack --version
> nstack 0.0.3
```

You can find the list of commands and options available by running

```
$ nstack --help
```

Commands

This section explains the commands supported by the CLI toolkit.

`register`

```
$ nstack register username email [-s SERVER]
```

Option	Description
`username`	A unique username to assign on the the server.
`email`	An email address to validate the user and send login credentials to.
`SERVER`	The remote NStack Server to register with, by default this will use our demo server.

A simple command to register with a remote NStack server so you can login, build modules, start workflows, etc. Upon successful registration you will receive credentials over email that you can paste into the `nstack` CLI and get started.

`set-server`

```
$ nstack set-server server-addr server-port id key
```

This command configures your local NStack CLI to communicate with a remote NStack Server with which you have registered (see previous command). You usually don't have to enter this command by hand, it will be contained with an email after successful registration that you can paste directly into your terminal.

Internally this modifies the `nstack.conf` CLI config file on your behalf (found in `~\.config` on Linux/macOS, and in `HOME/AppUser/Roaming` on Windows).

`info`

```
$ nstack info
```

Displays information regarding the entire current state of NStack, including:

- Modules
- Sources
- Sinks
- Running processes
- Base images

`init`

```
$ nstack init <stack>
```

Option	Description
`stack`	The default stack to use to build your service, e.g. `python` or `workflow` (`.nml` NStack Workflow Language).

Initialises a new nstack module in the current directory using the specified base language stack. This creates a working skeleton project which you can use to write your module.

If you are creating a module in an existing programming language, such as Python, `init` creates a module with a single `numChars` function already created. The initial project is comprised of the following files,

- `nstack.yaml`, your service's configuration file (see module_structure),
- `service.py`, an application file (or service.js, etc.), where your business-logic lives
- an empty packages file (e.g. `requirements.txt` for Python, or `package.json` for Node, etc.).

`init` is the command used to create a new workflow. In this case, NStack creates a skeleton `workflow.nml` file.

build

```
$ nstack build
```

Builds a module on your hosted nstack instance.

Note: `build` is also used to build workflows. Remember, workflows are modules too!

start

```
$ nstack start <workflow>
```

Option	Description
`<workflow>`	The workflow to start, in NStack Workflow Language

Used to start a workflow as a process. Workflows can either be provided as an argument such as:

Or, if you have built a workflow as a module, you can start it with:

notebook

```
$ nstack notebook
```

Create an interactive session within the terminal that provides a mini-REPL (you can also redirect a file/stream into the notebook command to provide for rapid service testing and development).

From this command-line, you can import modules as needed, and enter a single workflow that will be compiled and run immediately on the server (press `<Ctrl-D>` on Linux/macOS or `<Ctrl-Z>` on Windows to submit your input).

```
$ nstack notebook
import Demo.Classify:0.0.3 as D;
Sources.http<Text> { http_path = "/classify" } | D.numChars | Sinks.log<Text>
<Ctrl-D>
> Service started successfully as process 5
```

`send`

```
$ nstack send "route" 'data'
```

Option	Description
`route`	The endpoint to send the data where a workflow is running.
`data`	A json snippet to send to the endpoint and pass into a workflow.

Used with the HTTP source, `nstack send` sends a JSON-encoded element to an endpoint on the NStack server where a workflow has been started. Useful for testing workflows that are to be used as web-hooks.

`ps`

```
$ nstack ps
```

Shows a list of all processes, which are workflows that are running on your your nstack server.

`stop`

```
$ nstack stop <process-id>
```

Stop a running process.

`list`

```
$ nstack list <primitive>
```

Option	Description
`<primitive>`	The primitive you want to list.

Shows a list of available primitives. Support primitives are modules, workflows, functions, sources, and sinks.

`delete`

```
$ nstack delete <module>
```

Deletes a module (and thus its functions) from NStack.

`log`

```
$ nstack log <process>
```

Option	Description
`<process>`	The id of the process.

View the logs of a running process.

`server-logs`

```
$ nstack server-logs
```

View the full logs of the NStack server.

`gc`

```
$ nstack gc
```

Expert: Garbage-collect unused module images to free up space on the server.

Module Structure

Introduction

The following files are created when you create an nstack module using:

```
$ nstack init <stack>
```

nstack.yaml

The `nstack.yaml` file describes the configuration of your nstack module. It has several required fields that describe the project and let you control the packaging of your module.

Sample `nstack.yaml` file:

```yaml
# Module name
name: MyModule

# The language stack to use
stack: python

# Parent Image
parent: NStack.Python:0.25.0

api: |
  numChars : Text -> Integer

# (Optional) System-level packages needed
packages: []

# (Optional) Commands to run when building the module (Bash-compatible)
commands: []

# (Optional) Files/Dir to copy across into the module (can use regex/glob syntax)
files: []
```

`name`

Required

The name of the module. This is created by *nstack init* and uses the name of the current directory by default.

`stack`

Required

The base language stack to use when creating an image. Currently we support:

Name	Description
python	Python 3

`parent`

Required

The base-image your module builds from. This is typically generated by default when you create a module, but can be specified to include custom base-images which include standardised packages.

`api`

Required

The API which defines which functions you want to expose, and their input and output schemas. Please see idl_language for more details.

`files`

Optional

A list of files and directories within the project directory to include and bundle in alongside the image.

`packages`

Optional

A list of operating systems packages your module requires. These can be any packages installable via `dnf` on RHEL or Fedora.

Supported Types

Primitive types

NStack supports the following primitive types:

- `Integer`
- `Double`
- `Boolean`
- `Text`

More complex types can be built out of primitive ones:

- Optional types: `type1 optional`
- Tuples: `(type1, type2, ...)`. A tuple must have at least two fields.
- Structs: `{ name1: type1, name2: type2, ... }`
- Arrays: `[type1]`
- Sums: `Name1 type1a ... | Name2 type2a ... | ...`

A user can define their own type in the idl_language.

Workflow Language

Overview

A module consists of a module header, import statements, and definitions, for instance:

```
module ModuleB:0.1.0 {
  import ModuleA:0.1.0 as A;
  def x = A.y | A.z;
}
```

An import statement includes the module to be imported (`MyModule:0.1.0`) and its alias (`A`). The alias is used to qualify functions imported from the module, e.g. `A.y`.

Definitions bind function names (`x`) to expressions (`A.y | A.z`). Any function `x` defined in a module `ModuleB` can be used in any other module:

```
import ModuleB:0.1.0 as B;
def z = filter B.x;
```

If a name is not prefixed by a module alias, it refers to a function defined in the current module.

Expressions

Expressions combine already defined functions through the following operations:

Pipe

`A.y | A.z`

Every value produced by `A.y` is passed to `A.z`.

The output type of `A.y` must match the input type of `A.z`.

Concat

`concat A.y` or `A.y*`

`A.y` must be a function that produces lists of values, in which case `concat A.y` is a function that "unpacks" the lists and yields the same values one by one.

Filter

`filter A.y` or `A.y?`

`A.y` must be a function that produces "optional" (potentially missing) values, in which case `filter A.y` is a function that filters out missing values.

Type application

`Sources.A<T>`

Some functions (notably, most sources and sinks) can be specialized to multiple input or output types. This is done with type application: `Sources.http<Text>` specializes `Sources.http` to the type `Text`.

Parameter application

`A.y { one = "...", two = "..." }.`

Parameters are analogous to UNIX environment variables in the following ways:

1. Parameters are inherited. E.g. in

```
y = x;
z = y { foo = "bar" };
```

   both functions `x` and `y` will have access to `foo` when `z` is called.

2. Parameters can be overridden. E.g. in

```
y = x { foo = "baz" };
z = y { foo = "bar" };
```

   `y` overrides the value of `foo` that is passed to `x`. Therefore, `x` will see the value of `foo` as `baz`, not `bar`.

Parameters are used to configure sources and sinks — for instance, to specify how to connect to a PostgreSQL database.

Parameters can also be used to configure user-defined modules. Inside a Python nstack method, the value of parameter `foo` can be accessed as `self.args["foo"]`.

Comments

The workflow language supports line and block comments. Line comments start with `//` and extend until the end of line. Block comments are enclosed in `/*` and `*/` and cannot be nested.

EBNF grammar

The syntax is defined in EBNF (ISO/IEC 14977) in terms of tokens.

```
module = 'module', module name, '{', {import}, {definition}, '}';
import = 'import', module name, 'as', module alias, ';';
definition = 'def', name, '=', expression, ';';
expression = expression1, {'|', expression1};
expression1 = application, '*'
            | application, '?'
            | 'concat', application
            | 'filter', application
            ;
application = term [arguments];
arguments = '{', argument binding, {',', argument binding}, '}';
argument binding = name, '=', literal;
term = '(', expression, ')'
     | qualified name ['<', type, '>']
     ;
```

IDL Language

An IDL block describes the interface of an external module (which could be written in Python, for instance). It is embedded in nstack.yaml

The IDL block consists of two sections: the first section declares types, the second section declares functions.

Types are declared using the `type` keyword:

```
type PlantInfo = { petalLength : Double
                 , petalWidth : Double
                 , sepalLength : Double
                 , sepalWidth : Double
                 }
type PlantSpecies = Text
```

The left-hand side of a type declaration is the new type name; the right-hand side must be an existing type. Type declarations don't need to be delimited in any way.

Function declarations have form `name : Type` and must be delimited by commas, for instance

```
gotham : MovieRecordImage -> MovieRecordImage,
kelvin : MovieRecordImage -> MovieRecordImage,
lomo : MovieRecordImage -> MovieRecordImage
```

Thus, the full IDL block might look like this:

```
type PlantInfo = { petalLength : Double
                 , petalWidth : Double
                 , sepalLength : Double
                 , sepalWidth : Double
                 }
type PlantSpecies = Text

gotham : MovieRecordImage -> MovieRecordImage,
kelvin : MovieRecordImage -> MovieRecordImage,
lomo : MovieRecordImage -> MovieRecordImage
```

Supported Integrations

NStack is built to integrate with existing infrastructure, event, and data-sources. Typically, this is by using them as *sources* and *sinks* in the NStack Workflow Language.

See also:

Learn more about *sources* and *sinks* in *Concepts*

Sources

Postgres

```
Sources.postgres<Text> {
  pg_host = "localhost", pg_port = "5432",
  pg_user = "user", pg_password = "123456",
  pg_database = "db", pg_query = "SELECT * FROM tbl;" }
```

`pg_port` defaults to 5432, `pg_user` defaults to `postgres`, and `pg_password` defaults to the empty string. The other parameters are mandatory.

HTTP

```
Sources.http<Text> { http_path = "/foo" }
```

RabbitMQ (AMQP)

```
Sources.amqp<Text> {
  amqp_host = "localhost", amqp_port = "5672",
  amqp_vhost = "/", amqp_exchange = "ex",
  amqp_key = "key"
}
```

`amqp_port` defaults to 5672 and `amqp_vhost` defaults to `/`. The other parameters are mandatory.

Sinks

Postgres

```
Sinks.postgres<Text> {
  pg_host = "localhost", pg_port = "5432",
  pg_user = "user", pg_password = "123456",
  pg_database = "db", pg_table = "tbl" }
```

Like for Postgres source, `pg_port` defaults to 5432, `pg_user` defaults to `postgres`, and `pg_password` defaults to the empty string. The other parameters are mandatory.

NStack Log

```
Sinks.log<Text>
```

The Log sink takes no parameters.

RabbitMQ (AMQP)

```
Sinks.amqp<Text> {
  amqp_host = "localhost", amqp_port = "5672",
  amqp_vhost = "/", amqp_exchange = "ex",
  amqp_key = "key"
}
```

Like for AMQP source, `amqp_port` defaults to 5672 and `amqp_vhost` defaults to `/`. The other parameters are mandatory.