Twined

Release x.y.unknown

Octue Ltd

Nov 20, 2020

CONTENTS

Attention: This library is in very early stages. Like the idea of it? Please star us on GitHub and contribute via the issues board and roadmap.

twined is a library to help create and connect *Digital Twins* and data services.

> *"Twined" [t-why-nd] ~ encircled, twisted together, interwoven*

A digital twin is a virtual representation of a real life being - a physical asset like a wind turbine or car - or even a human. Like real things, digital twins need to interact, so can be connected together, but need a common communication framework to do so.

twined helps you to define a single file, a "twine", that defines a digital twin / data service. It specifies specifying its data interfaces, connections to other twins, and other requirements.

Any person, or any computer, can read a twine and understand *what-goes-in* and *what-comes-out*. That makes it easy to collaborate with other teams, since everybody is crystal clear about what's needed.

Fig. 1: Digital twins / data services connected in a hierarchy. Each blue circle represents a twin, coupled to its neighbours. Yellow nodes are where schema are used to connect twins.

CHAPTER

ONE

AIMS

twined provides a toolkit to help create and validate "twines" - descriptions of a digital twin, what data it requires, what it does and how it works.

The goals of this twined library are as follows:

- Provide a clear framework for what a *twine* can and/or must contain
- Provide functions to validate incoming data against a known *twine*
- Provide functions to check that a *twine* itself is valid
- Provide (or direct you to) tools to create *twines* describing what you require

In *Anatomy Of The Twine File*, we describe the different parts of a twine (examining how digital twins connect and interact… building them together in hierarchies and networks). But you may prefer to dive straight in with the *Quick Start* guide.

The scope of **twined** is not large. Many other libraries will deal with hosting and deploying digital twins, still more will deal with the actual analyses done within them. **twined** purely deals with parsing and checking the information exchanged.

CHAPTER

TWO

RAISON D'ETRE

Octue believes that a lynchpin of solving climate change is the ability for all engineering, manufacturing, supply chain and infrastructure plant to be connected together, enabling strong optimisation and efficient use of these systems.

To enable engineers and scientists to build, connect and run digital twins in large networks (or even in small teams!) it is necessary for everyone to be on the same page - the *Gemini Principles* are a great way to start with that, which is why we've released this part of our technology stack as open source, to support those principles and help develop a wider ecosystem.

The main goal is to **help engineers and scientists focus on doing engineering and science** - instead of apis, data cleaning/management, and all this cloud-pipeline-devops-test-ci-ml BS that takes up 90% of a scientist's time, when they should be spending their valuable time researching migratory patterns of birds, or cell structures, or wind turbine performance, or whatever excites them.

CHAPTER

THREE

USES

At Octue, **twined** is used as a core part of our application creation process:

- As a format to communicate requirements to our partners in research projects
- As a tool to validate incoming data to digital twins
- As a framework to help establish schema when designing digital twins
- As a source of information on digital twins in our network, to help map and connect twins together

We'd like to hear about your use case. Please get in touch!

We use the GitHub Issue Tracker to manage bug reports and feature requests. Please note, this is not a "general help" forum; we recommend Stack Overflow for such questions. For really gnarly issues or for help designing digital twin schema, Octue is able to provide application support services for those building digital twins using **twined**.

3.1 Quick Start

3.1.1 Installation

twined is available on pypi, so installation into your python virtual environment is dead simple:

```
pip install twined
```

Don't have a virtual environment with pip? You probably should! `pyenv` is your friend. Google it.

Compilation

There is presently no need to compile **twined**, as it's written entirely in python.

Third party library installation

twined is for python >= 3.6 so expects that. Other dependencies can be checked in `setup.py`, and will automatically installed during the installation above.

Third party build requirements

> **Attention:**
>
> **Woohoo! There are no crazy dependencies that you have to compile and build for your particular system.** (you know the ones… they never *actually* compile, right?). We aim to keep it this way.

3.1.2 Create your first twine

Let's say we want a digital twin that accepts two values, uses them to make a calculation, then gives the result. Anyone connecting to the twin will need to know what values it requires, and what it responds with.

First, create a blank text file, call it *twine.json*. We'll give the twin a title and description. Paste in the following:

```
{
        "title": "My first digital twin... of an atomising discombobulator",
        "description": "A simple example... estimates the `foz` value of an atomising␣
↪discombobulator."
}
```

Now, let's define an input values strand, to specify what values are required by the twin. For this we use a json schema (you can read more about them in *Introducing JSON Schema*). Add the `input_values` field, so your twine looks like this:

```
{
    "title": "My first digital twin",
    "description": "A simple example to build on..."
    "input_values_schema": {
        "$schema": "http://json-schema.org/2019-09/schema#",
        "title": "Input Values schema for my first digital twin",
        "description": "These values are supplied to the twin by another program␣
↪(often over a websocket, depending on your integration provider). So as these␣
↪values change, the twin can reply with an update.",
        "type": "object",
        "properties": {
            "foo": {
                    "description": "The foo value... speed of the discombobulator's␣
↪input bobulation module, in m/s",
                    "type": "number",
                    "minimum": 10,
                    "maximum": 500
            },
            "baz": {
                    "description": "The baz value... period of the discombobulator's␣
↪recombulation unit, in s",
                    "type": "number",
                    "minimum": 0,
                    "maximum": 1000
            }
        }
    }
}
```

Finally, let's define an output values strand, to define what kind of data is returned by the twin:

```
"output_values_schema": {
    "$schema": "http://json-schema.org/2019-09/schema#",
    "title": "Output Values schema for my first digital twin",
    "description": "The twin will output data that matches this schema",
    "type": "object",
    "properties": {
        "foz": {
                "description": "Estimate of the foz value... efficiency of the␣
↪discombobulator in %",
                "type": "number",
                "minimum": 10,
                "maximum": 500
        }
    }
}
```

3.1.3 Load the twine

twined provides a *Twine()* class to load a twine (from a file or a json string). The loading process checks the twine itself is valid. It's as simple as:

```
from twined import Twine

my_twine = Twine(source='twine.json')
```

3.1.4 Validate some inputs

Say we have some json that we want to parse and validate, to make sure it matches what's required for input values.

```
my_input_values = my_twine.validate_input_values(json='{"foo": 30, "baz": 500}')
```

You can read the values from a file too. Paste the following into a file named `input_values.json`:

```
{
  "foo": 30,
  "baz": 500
}
```

Then parse and validate directly from the file:

```
my_input_values = my_twine.validate_input_values(source="input_values.json")
```

Attention: LIBRARY IS UNDER CONSTRUCTION! WATCH THIS SPACE FOR MORE!

3.2 Anatomy Of The Twine File

The main point of **twined** is to enable engineers and scientists to easily (and rigorously) define a digital twin or data service.

This is done by adding a `twine.json` file to the repository containing your code. Adding a *twine* means you can:

- communicate (to you or a colleague) what data is required by this service
- communicate (to another service / machine) what data is required
- deploy services automatically with a provider like Octue.

To just get started building a *twine*, check out the *Quick Start*. To learn more about twines in general, see *About Twines*. Here, we describe the parts of a *twine* ("strands") and what they mean.

3.2.1 Strands

A *twine* has several sections, called *strands*. Each defines a different kind of data required (or produced) by the twin.

Strand	Describes the twin's requirements for...
Configuration Values	Data, in JSON form, used for configuration of the twin/service.
Configuration Manifest	Files/datasets required by the twin at configuration/startup
Input Values	Data, in JSON form, passed to the twin in order to trigger an analysis
Input Manifest	Files/datasets passed with Input Values to trigger an analysis
Output Values	Data, in JSON form, that will be produced by the twin (in response to inputs)
Output Manifest	Files/datasets that will be produced by the twin (in response to inputs)
Credentials	Credentials that are required by the twin in order to access third party services
Children	Other twins, access to which are required for this twin to function
Monitors	Visual and progress outputs from an analysis

Values-based Strands

The `configuration_values_schema`, `input_values_schema` and `output_values_schema` strands are *values-based*, meaning the data that matches these strands is in JSON form.

Each of these strands is a *json schema* which describes that data.

Configuration Values Strand

This strand is a `configuration_values_schema`, that is used to check validity of any `configuration_values` data supplied to the twin at startup.

The Configuration Values Strand is generally used to define control parameters relating to what the twin should do, or how it should operate.

For example, should it produce output images as low resolution PNGs or as SVGs? How many iterations of a fluid flow solver should be used? What is the acceptable error level on an classifier algorithm?

Input Values Strand

This strand is an `input_values_schema`, that is used to check validity of `input_values` data supplied to the twin at the beginning of an analysis task.

The Input Values Strand is generally used to define actual data which will be processed by the twin. Sometimes, it may be used to define control parameters specific to an analysis.

For example, if a twin cleans and detects anomalies in a 10-minute timeseries of 1Hz data, the `input_values` might contain an array of data and a list of corresponding timestamps. It may also contain a control parameter specifying which algorithm is used to do the detection.

Note: Depending on the way the twin is deployed (see *Deployment*), the `input_values` might come in from a web request, over a websocket or called directly from the command line or another library.

However they come, if the new `input_values` validate against the `input_values_schema` strand, then analysis can proceed.

Output Values Strand

This strand is an `output_values_schema`, that is used to check results (`output_values`) computed during an analysis. This ensures that the application wrapped up within the *twine* is operating correctly, and enables other twins/services or the end users to see what outputs they will get.

For example,if a twin cleans and detects anomalies in a 10-minute timeseries of 1Hz data, the `output_values` might contain an array of data interpolated onto regular timestamps, with missing values filled in and a list of warnings where anomalies were found.

Let's look at basic examples for twines containing each of these strands:

Configuration Values Strand

This *twine* contains an example `configuration_values_schema` with one control parameter.

Many more detailed and specialised examples are available in the GitHub repository

```
{
      "configuration_values_schema": {
              "title": "The example configuration form",
              "description": "The Configuration Values Strand of an example twine",
              "type": "object",
              "properties": {
                      "n_iterations": {
                              "description": "An example of an integer configuration␣
↪variable, called 'n_iterations'.",
                              "type": "integer",
                              "minimum": 1,
                              "maximum": 10,
                              "default": 5
                      }
              }
      }
}
```

Matching `configuration_values` data could look like this:

```
{
    "n_iterations": 8,
}
```

Input Values Strand

This *twine* contains an example `input_values_schema` with one input value, which marked as required.

Many more detailed and specialised examples are available in examples.

```
{
    "input_values_schema": {
        "title": "Input Values",
        "description": "The input values strand of an example twine, with a required␣
↪height value",
        "type": "object",
        "properties": {
            "height": {
                "description": "An example of an integer value called 'height'",
                "type": "integer",
                "minimum": 2
            }
        },
        "required": ["height"]
    },
```

Matching `input_values` data could look like this:

```
{
    "height": 13,
}
```

Output Values Strand

Stuff

Manifest-based Strands

Frequently, twins operate on files containing some kind of data. These files need to be made accessible to the code running in the twin, in order that their contents can be read and processed. Conversely, a twin might produce an output dataset which must be understood by users.

The `configuration_manifest`, `input_manifest` and `output_manifest` strands describe what kind of datasets (and associated files) are required / produced.

Note: Files are always contained in datasets, even if there's only one file. It's so that we can keep nitty-gritty file metadata separate from the more meaningful, higher level metadata like what a dataset is for.

Configuration Manifest Strand

This describes datasets/files that are required at startup of the twin / service. They typically contain a resource that the twin might use across many analyses.

For example, a twin might predict failure for a particular component, given an image. It will require a trained ML model (saved in a `*.pickle` or `*.json`). While many thousands of predictions might be done over the period that the twin is deployed, all predictions are done using this version of the model - so the model file is supplied at startup.

Input Manifest Strand

These files are made available for the twin to run a particular analysis with. Each analysis will likely have different input datasets.

For example, a twin might be passed a dataset of LiDAR `*.scn` files and be expected to compute atmospheric flow properties as a timeseries (which might be returned in the *output values* for onward processing and storage).

Output Manifest Strand

Files are created by the twin during an analysis, tagged and stored as datasets for some onward purpose. This strand is not used for sourcing data; it enables users or other services to understand appropriate search terms to retrieve datasets produced.

Describing Manifests

Manifest-based strands are a **description of what files are needed**, NOT a list of specific files or datasets. This is a tricky concept, but important, since services should be reusable and applicable to a range of similar datasets.

The purpose of the manifest strands is to provide a helper to a wider system providing datafiles to digital twins.

The manifest strands therefore use **tagging** - they contain a `filters` field, which should be valid Apache Lucene search syntax. This is a powerful syntax, whose tagging features allow us to specify incredibly broad, or extremely narrow searches (even down to a known unique result). See the tabs below for examples.

Note: Tagging syntax is extremely powerful. Below, you'll see how this enables a digital twin to specify things like:

"OK, I need this digital twin to always have access to a model file for a particular system, containing trained model data"

"Uh, so I need an ordered sequence of files, that are CSV files from a meteorological mast."

This allows **twined** to check that the input files contain what is needed, enables quick and easy extraction of subgroups or particular sequences of files within a dataset, and enables management systems to map candidate datasets to twins that might be used to process them.

Configuration Manifest Strand

Here we construct an extremely tight filter, which connects this digital twin to datasets from a specific system.

Show twine containing this strand

```
{
  // Manifest strands contain lists, with one entry for each required dataset
  "configuration_manifest": [
    {
      // Once the inputs are validated, your analysis program can use this key to␣
↪access the dataset
      "key": "trained_model",
      // General notes, which are helpful as a reminder to users of the service
      "purpose": "The trained classifier",
      // Issues a strict search for data provided by megacorp, containing *.mdl files␣
↪tagged as
      // classifiers for blade damage on system abc123
      "filters": "organisation: megacorp AND tags:(classifier AND damage AND␣
↪system:abc123) AND files:(extension:mdl)"
    }
  ]
}
```

Show a matching file manifest

```
{
  "id": "8ead7669-8162-4f64-8cd5-4abe92509e17",
  "datasets": [
    {
      "id": "7ead7669-8162-4f64-8cd5-4abe92509e17",
```

(continues on next page)

(continued from previous page)

```
      "name": "training data for system abc123",
      "organisation": "megacorp",
      "tags": "classifier, damage, system:abc123",
      "files": [
        {
          "path": "datasets/7ead7669/blade_damage.mdl",
          "cluster": 0,
          "sequence": 0,
          "extension": "csv",
          "tags": "",
          "posix_timestamp": 0,
          "id": "abff07bc-7c19-4ed5-be6d-a6546eae8e86",
          "last_modified": "2019-02-28T22:40:30.533005Z",
          "name": "blade_damage.mdl",
          "size_bytes": 59684813,
          "sha-512/256": "somesha"
        }
      ]
    }
  ]
}
```

Input Manifest Strand

Here we specify that two datasets (and all or some of the files associated with them) are required, for a service that cross-checks meteorological mast data and power output data for a wind farm.

Show twine containing this strand

```
{
  // Manifest strands contain lists, with one entry for each required dataset
  "input_manifest_filters": [
    {
      // Once the inputs are validated, your analysis program can use this key to␣
↪access the dataset
      "key": "met_mast_data",
      // General notes, which are helpful as a reminder to users of the service
      "purpose": "A dataset containing meteorological mast data",
      // Searches datasets which are tagged "met*" (allowing for "met" and
↪"meterological"), whose
      // files are CSVs in a numbered sequence, and which occur at a particular␣
↪location
      "filters": "tags:(met* AND mast) AND files:(extension:csv AND sequence:>=0) AND␣
↪location:10"
    },
    {
      "key": "scada_data",
      "purpose": "A dataset containing scada data",
      // The organisation: filter refines search to datasets owned by a particular␣
↪organisation handle
      "filters": "organisation: megacorp AND tags:(scada AND mast) AND␣
↪files:(extension:csv AND sequence:>=0)"
    }
  ],
}
```

Show a matching file manifest

```
{
  "id": "8ead7669-8162-4f64-8cd5-4abe92509e17",
  "datasets": [
    {
      "id": "7ead7669-8162-4f64-8cd5-4abe92509e17",
      "name": "meteorological mast dataset",
      "tags": "met, mast, wind, location:108346",
      "files": [
        {
          "path": "input/datasets/7ead7669/mast_1.csv",
          "cluster": 0,
          "sequence": 0,
          "extension": "csv",
          "tags": "",
          "posix_timestamp": 1551393630,
          "id": "abff07bc-7c19-4ed5-be6d-a6546eae8e86",
          "last_modified": "2019-02-28T22:40:30.533005Z",
          "name": "mast_1.csv",
          "size_bytes": 59684813,
          "sha-512/256": "somesha"
        },
        {
          "path": "input/datasets/7ead7669/mast_2.csv",
          "cluster": 0,
          "sequence": 1,
          "extension": "csv",
          "tags": "",
          "posix_timestamp": 1551394230,
          "id": "bbff07bc-7c19-4ed5-be6d-a6546eae8e45",
          "last_modified": "2019-02-28T22:50:40.633001Z",
          "name": "mast_2.csv",
          "size_bytes": 59684813,
          "sha-512/256": "someothersha"
        }
      ]
    },
    {
      "id": "5cf9e445-c288-4567-9072-edc31003b022",
      "name": "scada data exports",
      "tags": "wind, turbine, scada, system:ab32, location:108346",
      "files": [
        {
          "path": "input/datasets/7ead7669/export_1.csv",
          "cluster": 0,
          "sequence": 0,
          "extension": "csv",
          "tags": "",
          "posix_timestamp": 1551393600,
          "id": "78fa511f-3e28-4bc2-aa28-7b6a2e8e6ef9",
          "last_modified": "2019-02-28T22:40:00.000000Z",
          "name": "export_1.csv",
          "size_bytes": 88684813,
          "sha-512/256": "somesha"
        },
        {
          "path": "input/datasets/7ead7669/export_2.csv",
          "cluster": 0,
```


(continues on next page)

(continued from previous page)

```
          "sequence": 1,
          "extension": "csv",
          "tags": "",
          "posix_timestamp": 1551394200,
          "id": "204d7316-7ae6-45e3-8f90-443225b21226",
          "last_modified": "2019-02-28T22:50:00.000000Z",
          "name": "export_2.csv",
          "size_bytes": 88684813,
          "sha-512/256": "someothersha"
        }
      ]
    }
  ]
}
```

Output Manifest Strand

Output figure files (with *.fig extension) containing figures enabling a visual check of correlation between met mast and scada data.

Show twine containing this strand

```
{
  "output_manifest_filters": [
    {
      // Twined will prepare a manifest with this key, which you can add to during␣
↪the analysis or once its complete
      "key": "met_scada_checks",
      // General notes, which are helpful as a reminder to users of the service
      "purpose": "A dataset containing figures showing correlations between mast and␣
↪scada data",
      // Twined will check that the output file manifest has tags appropriate to the␣
↪filters
      "filters": "tags:(met* AND scada AND correlation) AND files:(extension:json)␣
↪AND location:*"
    }
  ]
}
```

Show a matching file manifest

```
{
  "id": "8ead7669-8162-4f64-8cd5-4abe92509e17",
  "datasets": [
    {
      "id": "4564deca-5654-42e8-aadf-70690b393a30",
      "name": "visual cross check data",
      "organisation": "megacorp",
      "tags": "figure, met, mast, scada, check, location:108346",
      "files": [
        {
          "path": "datasets/7ead7669/cross_check.fig",
          "cluster": 0,
          "sequence": 0,
          "extension": "fig",
          "tags": "",
          "posix_timestamp": 1551394800,
```

(continues on next page)

(continued from previous page)

```
            "id": "38f77fe2-c8c0-49d1-a08c-0928d53a742f",
            "last_modified": "2019-02-28T23:00:00.000000Z",
            "name": "cross_check.fig",
            "size_bytes": 59684813,
            "sha-512/256": "somesha"
        }
      ]
    }
  ]
}
```

TODO - clean up or remove this section

It's the job of **twined** to make sure of two things:

1. make sure the *twine* file itself is valid,

   File data (input, output)

   Files are not streamed directly to the digital twin (this would require extreme bandwidth in whatever system is orchestrating all the twins). Instead, files should be made available on the local storage system; i.e. a volume mounted to whatever container or VM the digital twin runs in.

   Groups of files are described by a `manifest`, where a manifest is (in essence) a catalogue of files in a dataset.

   A digital twin might receive multiple manifests, if it uses multiple datasets. For example, it could use a 3D point cloud LiDAR dataset, and a meteorological dataset.

```
{
    "manifests": [
        {
            "type": "dataset",
            "id": "3c15c2ba-6a32-87e0-11e9-3baa66a632fe",  //␣
↪UUID of the manifest
            "files": [
                {
                    "id": "abff07bc-7c19-4ed5-be6d-a6546eae8e86",␣
↪ // UUID of that file
                    "sha1": "askjnkdfoisdnfkjnkjsnd"  // for␣
↪quality control to check correctness of file contents
                    "name": "Lidar - 4 to 10 Dec.csv",
                    "path": "local/file/path/to/folder/containing/
↪it/",
                    "type": "csv",
                    "metadata": {
                    },
                    "size_bytes": 59684813,
                    "tags": "lidar, helpful, information, like,␣
↪sequence:1",  // Searchable, parsable and filterable
                },
                {
                    "id": "abff07bc-7c19-4ed5-be6d-a6546eae8e86",
                    "name": "Lidar - 11 to 18 Dec.csv",
                    "path": "local/file/path/to/folder/containing/
↪it/",
                    "type": "csv",
```

(continues on next page)

(continued from previous page)

```
                    "metadata": {
                    },
                    "size_bytes": 59684813,
                    "tags": "lidar, helpful, information, like,␣
↪sequence:2",  // Searchable, parsable and filterable
                },
                {
                    "id": "abff07bc-7c19-4ed5-be6d-a6546eae8e86",
                    "name": "Lidar report.pdf",
                    "path": "local/file/path/to/folder/containing/
↪it/",
                    "type": "pdf",
                    "metadata": {
                    },
                    "size_bytes": 484813,
                    "tags": "report",  // Searchable, parsable␣
↪and filterable
                }
            ]
        },
        {
            // ... another dataset manifest ...
        }
    ]
}
```

Credentials Strand

In order to:

- GET/POST data from/to an API,
- query a database, or
- connect to a socket (for receiving Values or emitting Values, Monitors or Logs),

A digital twin must have *access* to it. API keys, database URIs, etc must be supplied to the digital twin but treated with best practice with respect to security considerations. The purpose of the `credentials` strand is to dictate what credentials the twin requires in order to function.

Defining the Credentials Strand

This is the simplest of the strands, containing a list of credentials (whose `NAMES_SHOULD_BE_SHOUTY_SNAKE_CASE`) with a reminder of the purpose. Defaults can also be provided, useful for running on local or closed networks.

```
{
  "credentials": [
    {
      "name": "SECRET_THE_FIRST",
      "purpose": "Token for accessing a 3rd party API service"
    },
    {
      "name": "SECRET_THE_SECOND",
      "purpose": "Token for accessing a 3rd party API service"
```

(continues on next page)

(continued from previous page)

```
    },
    {
      "name": "SECRET_THE_THIRD",
      "purpose": "Usually a big secret but sometimes has a convenient non-secret␣
↪default, like a sandbox or local database",
      "default": "postgres://pguser:pgpassword@localhost:5432/pgdb"
    }
  ]
}
```

Supplying Credentials

Attention: *Credentials should never be hard-coded into application code*

Do you trust the twin code? If you insert credentials to your own database into a digital twin provided by a third party, you better be very sure that twin isn't going to scrape all that data out then send it elsewhere!

Alternatively, if you're building a twin requiring such credentials, it's your responsibility to give the end users confidence that you're not abusing their access.

There'll be a lot more discussion on these issues, but it's outside the scope of **twined** - all we do here is make sure a twin has the credentials it requires.

Credentials should be securely managed by whatever system is managing the twin, then made accessible to the twin in the form of environment variables:

```
SERVICE_API_
↪KEY=someLongTokenTHatYouProbablyHaveToPayTheThirdPartyProviderLoadsOfMoneyFor
```

Credentials may also reside in a `.env` file in the current directory, either in the format above (with a new line for each variable) or, for convenience, as bash exports like:

```
export SERVICE_API_
↪KEY=someLongTokenTHatYouProbablyHaveToPayTheThirdPartyProviderLoadsOfMoneyFor
```

The `validate_credentials()` method of the `Twine` class checks for their presence and, where contained in a `.env` file, ensures they are loaded into the environment.

Monitors Strand

The `configuration_values_schema`, `input_values_schema` and `output_values_schema` strands are *values-based*, meaning the data that matches these strands is in JSON form.

Each of these strands is a *json schema* which describes that data.

Monitors Strand

There are two kinds of monitoring data required from a digital twin.

Monitor data (output)

Values for health and progress monitoring of the twin, for example percentage progress, iteration number and status - perhaps even residuals graphs for a converging calculation. Broadly speaking, this should be user-facing information.

This kind of monitoring data can be in a suitable form for display on a dashboard

Log data (output)

Logged statements, typically in iostream form, produced by the twin (e.g. via python's `logging` module) must be capturable as an output for debugging and monitoring purposes. Broadly speaking, this should be developer-facing information.

Let's look at basic examples for twines containing each of these strands:

Monitors Strand

Monitor data (output)

Log data (output)

Children Strand

> **Attention:** Coming Soon!

3.2.2 Twine File Schema

Because the `twine.json` file itself is in `JSON` format with a strict structure, **twined** uses a schema to make that twine files are correctly written (a "schema-schema", if you will, since a twine already contains schema). Try not to think about it. But if you must, the *twine* schema is here.

The first thing **twined** always does is check that the `twine.json` file itself is valid, and give you a descriptive error if it isn't.

3.2.3 Other External I/O

A twin might:

- GET/POST data from/to an external API,
- query/update a database,
- upload files to an object store,
- trigger events in another network, or
- perform pretty much any interaction you can think of with other applications over the web.

However, such data exchange may not be controllable by **twined** (which is intended to operate at the boundaries of the twin) unless the resulting data is returned from the twin (and must therefore be compliant with the schema).

So, there's nothing for **twined** to do here, and no need for a strand in the *twine* file. However, interacting with third party APIs or databases might require some credentials. See *Credentials Strand* for help with that.

Note: This is actually a very common scenario. For example, the purpose of the twin might be to fetch data (like a weather forecast) from some external API then return it in the `output_values` for use in a network of digital twins. But its the twin developer's job to do the fetchin' and make sure the resulting data is compliant with the `output_values_schema` (see *Values-based Strands*).

3.3 About Twines

Twined is a framework for describing a digital twin or data service.

We call these descriptions "twines". To just get started building a *twine*, check out the *Quick Start*. To get into the detail of what's in a *twine*, see *Anatomy Of The Twine File*.

Here, we look at requirements for the framework, our motivations and background, and some of the decisions made while developing **twined**.

3.3.1 Digital Twins

A digital twin is a virtual representation of a real life being - a physical asset like a wind turbine or car - or even a human.

There are three reasons why you might want to create a digital twin:

- Monitoring
- Prediction
- Optimisation

On its own, a digital twin can be quite useful. For example, a twin might embody an AI-based analysis to predict power output of a turbine.

Fig. 1: A digital twin consists of some kind of analysis or processing task, which could be run many times per second, or daily, down to occasionally or sometimes only once (the same as a "normal" analysis).

Coupling digital twins is generally even more useful. You might wish to couple your turbine twin with a representation of the local power grid, and a representation of a factory building to determine power demand... enabling you to optimise your factory plant for lowest energy cost whilst intelligently selling surplus power to the grid.

Fig. 2: A hierarchy of digital twins. Each blue circle represents a twin, coupled to its neighbours. Yellow nodes are where schema are used to connect twins.

Gemini Principles

The Gemini Principles have been derived by the Centre for Digital Built Britain (CDBB). We strongly recommend you give them a read if embarking on a digital twins project.

The aim of **twined** is to enable the following principles. In particular:

1. Openness (open-source project to create schema for twins that can be run anywhere, anywhen)
2. Federation (encouraging a standardised way of connecting twins together)
3. Security (making sure schemas and data can be read safely)
4. Public Good (see our nano-rant about climate change in *Raison d'etre*)

3.3.2 Requirements of the framework

A *twine* must describe a digital twin, and have multiple roles. It must:

1. Define what data is required by a digital twin, in order to run
2. Define what data will be returned by the twin following a successful run
3. Define the formats of these data, in such a way that incoming data can be validated
4. Define what other (1st or 3rd party) twins / services are required by this one in order for it to run.

If this weren't enough, the description:

1. Must be trustable (i.e. a *twine* from an untrusted, corrupt or malicious third party should be safe to at least read)
2. Must be machine-readable *and machine-understandable*[1]
3. Must be human-readable *and human-understandable*[1]
4. Must be discoverable (that is, searchable/indexable) otherwise people won't know it's there in orer to use it.

Fortunately for digital twin developers, several of these requirements have already been seen for data interchange formats developed for the web. **twined** uses `JSON` and `JSONSchema` to help interchange data.

If you're not already familiar with `JSONSchema` (or wish to know why **twined** uses `JSON` over the seemingly more appropriate `XML` standard), see *Introducing JSON Schema*.

3.3.3 Introducing JSON Schema

`JSON` is a data interchange format that has rapidly taken over as the defacto web-based data communication standard in recent years.

`JSONSchema` is a way of specifying what a `JSON` document should contain. The Schema are, themselves, written in `JSON`!

Whilst schema can become extremely complicated in some scenarios, they are best designed to be quite succinct. See below for the schema (and matching `JSON`) for an integer and a string variable.

JSON:

```
{
    "id": 1,
    "name": "Tom"
}
```

Schema:

```
{
    "type": "object",
    "title": "An id number and a name",
    "properties": {
      "id": {
        "type": "integer",
        "title": "An integer number",
        "default": 0
      },
      "name": {
        "type": "string",
```

(continues on next page)

[1] *Understandable* essentially means that, once read, the machine or human knows what it actually means and what to do with it.

(continued from previous page)

```
        "title": "A string name",
        "default": ""
      }
    }
}
```

Useful resources

Link	Resource
https://jsonschema.net/	Useful web tool for inferring schema from existing json
https://jsoneditoronline.org	A powerful online editor for json, allowing manipulation of large documents better than most text editors
https://www.json.org/	The JSON standard spec
https://json-schema.org/	The (draft standard) JSONSchema spec
https://rjsf-team.github.io/react-jsonschema-form/	A front end library for generating webforms directly from a schema

Human readability

Back in our *Requirements of the framework* section, we noted it was important for humans to read and understand schema.

The actual documents themselves are pretty easy to read by technical users. But, for non technical users, readability can be enhanced even further by the ability to turn `JSONSchema` into web forms automatically. For our example above, we can autogenerate a web form straight from the schema:

Fig. 3: Web form generated from the example schema above.

Thus, we can take a schema (or a part of a schema) and use it to generate a control form for a digital twin in a web interface without writing a separate form component - great for ease and maintainability.

Why not XML?

In a truly excellent three-part blog, writer Seva Savris takes us through the ups and downs of JSON versus XML; well worth a read if wishing to understand the respective technologies better.

In short, both JSON and XML are generalised data interchange specifications and can both can do what we want here. We choose JSON because:

1. Textual representation is much more concise and easy to understand (very important where non-developers like engineers and scientists must be expected to interpret schema)
2. Attack vectors. Because entities in XML are not necessarily primitives (unlike in JSON), an XML document parser in its default state may leave a system open to XXE injection attacks and DTD validation attacks, and therefore requires hardening. JSON documents are similarly afflicated (just like any kind of serialized data) but default parsers, operating on the premise of only deserializing to primitive types, are safe by default - it is only when nondefault parsering or deserialization techniques (such as JSONP) are used that the application becomes vulnerable. By utilising a default JSON parser we can therefore significantly shrink the attack surface of the system. See this blog post for further discussion.
3. XML is powerful... perhaps too powerful. The standard can be adapted greatly, resulting in high encapsulation and a high resilience to future unknowns. Both beneficial. However, this requires developers of twins to maintain interfaces of very high complexity, adaptable to a much wider variety of input. To enable developers to progress, we suggest handling changes and future unknowns through well-considered versioning, whilst keeping their API simple.
4. XML allows baked-in validation of data and attributes. Whilst advantageous in some situations, this is not a benefit here. We wish validation to be one-sided: validation of data accepted/generated by a digital twin should be occur within (at) the boundaries of that twin.
5. Required validation capabilities, built into XML are achievable with JSONSchema (otherwise missing from the pure JSON standard)
6. JSON is a more compact expression than XML, significantly reducing memory and bandwidth requirements. Whilst not a major issue for most modern PCS, sensors on the edge may have limited memory, and both memory and bandwidth at scale are extremely expensive. Thus for extremely large networks of interconnected systems there could be significant speed and cost savings.

3.3.4 Other Considerations

A variety of thoughts that arose whilst architecting **twined**.

Bash-style stdio

Some thought was given to using a very old-school-unix approach to piping data between twins, via stdout.

Whilst attractive (as being a wildly fast way of piping data between twins on the same machine) it was felt this was insufficiently general, eg:

- where twins don't exist on the same machine or container, making it cumbersome to engineer common iostreams
- where slight differences between different shells might lead to incompatibilities or changes in behaviour

And also unfriendly, eg:

- engineers or scientists unfamiliar with subtleties of bash shell scripting encounter difficulty piping data around
- difficult to build friendly web based tools to introspect the data and configuration
- bound to be headaches on windows platforms, even though windows now supports bash

- easy to corrupt using third party libraries (e.g. which print to stdout)

Units

Being used (mostly) for engineering and scientific analysis, it was tempting to add in a specified sub-schema for units. For example, mandating that where values can be given in units, they be specified in a certain way, like:

```
{
    "wind_speed": {
        "value": 10.2,
        "units": "mph"
    }
}
```

or (more succinct):

```
{
    "wind_speed": 10.2,
    "wind_speed_units": "mph"
}
```

It's still extremely tempting to provide this facility; or at least provide some way of specifying in the schema what units a value should be provided in. Thinking about it but don't have time right now. If anybody wants to start crafting a PR with an extension or update to **twined** that facilitates this; please raise an issue to start progressing it.

Variable Style

A premptive stamp on the whinging...

Note that in the `JSON` descriptions above, all variables are named in `snake_case` rather than `camelCase`. This decision, more likely than even Brexit to divide opinions, is based on:

- **The languages we anticipate being most popular for building twins seem to trend toward snake case (eg** python, c++) although to be fair we might've woefully misjudged which languages start emerging.
- **The reservation of snake case for the schema spec has the subtle advantage that in future, we might be able to use** camelCase within the spec to denote class types in some useful way, just like in python. Not sure yet; just mulling.
- **The *Requirements of the framework* mention human-readability as a must;** this paper suggests a 20% slower comprehension of camel case than snake, although to be fair that's probably arguable.
- **We're starting in Python so are taking a lead from PEP8, which is bar none the most successful style guide on the** planet, because it got everybody on the same page really early on.

If existing code that you're dropping in uses camelCase, please don't file that as an issue... converting property names automatically after schema validation generation is trivial, there are tons of libraries (like humps) to do it.

We'd also consider a pull request for a built-in utility converting to and from that does this following validation and prior to returning results. Suggest your proposed approach on the issues board.

Language Choice

twined is presently released in python only. It won't be too hard to replicate functionality in other languages, and we're considering other languages at present, so might be easily persuadable ;)

If you require implementation of **twined** in a different language, and are willing to consider sponsorship of development and maintenance of that library, please file an issue.

3.4 Deployment

3.4.1 Deploying with Octue

Octue provides automated deployment to a cloud provider (like GCP or Azure), along with permissions and user management, monitoring, logging and data storage management out of the box.

There are also a whole bunch of collaborative helper tools, like the graphical twine builder and manifesting tools, designed to speed up the process of building and using twines.

The full set of services is in early beta, get in touch and we can help you architect systems - from small data services to large networks of *Digital Twins*.

3.4.2 Coming Soon - Deploying with doctue

Once we've bedded down our services internally at Octue, we'll be open-sourcing more parts of our build/deploy process, including docker containers with pre-configured servers to run and monitor twine-based services and digital twins.

This will allow services to be easily spun up on GCP, Azure Digital Ocean etc., and be a nice halfway house between fully managed system on Octue and running your own webserver. Of course, without all the collaborative and data management features that Octue provides ;)

We're looking for commercial sponsors for this part of the process - if that could be you, please get in touch

3.4.3 Deploying as a command-line application

Use the open-source octue app template as a guide. Write your new python code (or call your existing tools/libraries) within it. It's set up to wrap and check configuration, inputs and outputs using twined. Follow the instructions there to set up your inputs, and your files, and run an analysis.

3.4.4 Deploying with your own web server

You can use any python based web server (need another language? see *Language Choice*):

- Add `configuration_values_data` to your webserver config
- Set up an endpoint to allow.
- Set up an endpoint to handle incoming requests / socket messages - these will be `input_values_data`.
- Treat these requests / messages as events which trigger a task.
- **In your task framework (e.g. your celery task), either:**

  - Use **twined** directly to validate the `input_values_data`/`output_values_data` (and, on startup, the `configuration_values_data`) and handle running any required analysis yourself, or
  - import your analysis app (as built in *Deploying as a command-line application*) and call it with the configuration and input data in your task framework.
- Return the result to the client.

3.5 License

Octue maintains **twined** as an open source project, under the MIT license.

3.5.1 The boring bit

3.5.2 Third Party Libraries

twined includes or is linked against the following third party libraries:

Plotly.js

OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

jsonschema

Copyright (c) 2013 Julian Berman

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

3.6 Version History

3.6.1 Origins

twined began as an internal tool at Octue, enabling applications to be connected together in the Octue ecosystem.

The twined library is presently being ported out of Octue's SDKs as it became clear that it would be most beneficial to open-source the framework we developed to connect applications and digital twins together.

3.6.2 0.0.x

Initial library framework - development version. Highly unstable! Let's see what happens...

New Features

1. Documentation
2. Travis- and RTD- based test and documentation build with Codecov integration
3. Load and validation of twine itself against twine schema
4. Main `Twine()` class with strands set as attributes
5. Validation of input, config and output values against twine
6. Validation of manifest json
7. Credential parsing from the environment and validation
8. Hook allowing instantiation of inputs and config to a given class e.g. `Manifest`
9. Tests to cover the majority of functionality

Backward Incompatible API Changes

1. n/a (Initial release)

Bug Fixes & Minor Changes

1. n/a (Initial Release)