Twined

Release x.y.unknown

Tom Clark

Jan 10, 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 *Digital Twins* talk to one another.

> *"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, 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*.

Fig. 1: Digital twins 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 1

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 twined are as follows:

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

In *About Twines (Schema)*, 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 2

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.

CHAPTER 3

Uses

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

- 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**.

CHAPTER 4

Life Choices

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 get in touch.

4.1 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.

4.1.1 Compilation

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

4.1.2 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.

4.1.3 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.

4.2 Quick Start

4.2.1 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
        }
    }
}
```

4.2.2 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 is valid. It's as simple as:

```
from twined import Twine

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

4.2.3 Validate some inputs

Attention: LIBRARY IS UNDER CONSTRUCTION! WATCH THIS SPACE!

4.3 Examples

Here, we look at example use cases for the library, and show how to use it in python.

It's also well worth looking at the unit test cases copied straight from the unit test cases, so you can always check there to see how everything hooks up.

4.3.1 [Simple] Equipment installation cost

Scenario

You need to provide your team with an estimate for installation cost of an equipment foundation.

It's a straightforward calculation for you, but the Logistics Team keeps changing the installation position, to try and optimise the overall project logistics.

Each time the locations change, the GIS team gives you an updated embedment depth, which is what you use (along with steel cost and foundation type), to calculate cost and report it back.

This twine allows you to define to create a wrapper around your scripts that communicates to the GIS team what you need as an input, communicate to the logistics team what they can expect as an output.

When deployed as a digital twin, the calculation gets automatically updated, leaving you free to get on with all the other work!

Twine

We specify the `steel_cost` and `foundation_type` as `configuration` values, which you can set on startup of the twin.

Once the twin is running, it requires the `embedment_depth` as an `input_value` from the GIS team. A member of the GIS team can use your twin to get `foundation_cost` directly.

```
{
    "title": "Foundation Cost Model",
    "description": "This twine helps compute the cost of an installed foundation.",
    "children": [
    ],
    "configuration_schema": {
        "$schema": "http://json-schema.org/2019-09/schema#",
        "title": "Foundation cost twin configuration",
        "description": "Set config parameters and constants at startup of the twin.",
        "type": "object",
        "properties": {
            "steel_cost": {
                "description": "The cost of steel in GBP/m^3. To get a better␣
↪predictive model, you could add an economic twin that forecasts the cost of steel␣
↪using the project timetable.",
                "type": "number",
                "minimum": 0,
                "default": 3000
            },
            "foundation_type": {
                "description": "The type of foundation being used.",
                "type": "string",
                "pattern": "^(monopile|twisted-jacket)$",
                "default": "monopile"
            }
        }
    },
    "input_values_schema": {
        "$schema": "http://json-schema.org/2019-09/schema#",
        "title": "Input Values schema for the foundation cost twin",
        "description": "These values are supplied to the twin asynchronously over a␣
↪web socket. So as these values change, the twin can reply with an update.",
        "type": "object",
        "properties": {
            "embedment_depth": {
                "description": "Embedment depth in metres",
                "type": "number",
                "minimum": 10,
                "maximum": 500
            }
        }
    },
    "output_manifest": [
    ],
    "output_values_schema": {
        "title": "Output Values schema for the foundation cost twin",
        "description": "The response supplied to a change in input values will always␣
↪conform to this schema.",
```

(continues on next page)

(continued from previous page)

```
        "type": "object",
        "properties": {
            "foundation_cost": {
                "description": "The foundation cost.",
                "type": "integer",
                "minimum": 2
            }
        }
    }
}
```

4.4 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.

4.4.1 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*)

4.5 About Twines (Schema)

The core of **twined** is to provide and use schemas for digital twins.

Below, we set out requirements and a framework for creating a *schema* to represent a digital twin. We call these schema "twines". To just get started building a **twine**, check out the _quick_start.

4.5.1 Requirements of digital twin schema

A *schema* defines a digital twin, and has multiple roles. It:

1. Defines what data is required by a digital twin, in order to run
2. Defines what data will be returned by the twin following a successful run
3. Defines the formats of these data, in such a way that incoming data can be validated

If this weren't enough, the schema:

1. Must be trustable (i.e. a schema 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 searchable/indexable

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

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*.

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",
```

(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)

```
    "properties": {
      "id": {
        "type": "integer",
        "title": "An integer number",
        "default": 0
      },
      "name": {
        "type": "string",
        "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 digital twin schema* 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:

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

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

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.

4.5.2 Data framework

We cannot simply expect many developers to create digital twins with some schema, then to be able to connect them all together - even if those schema are all fully valid (*readable*). **twined** makes things slightly more specific.

twined has an opinionated view on how incoming data is organised. This results in a top-level schema that is extremely prescriptive (*understandable*), allowing digital twins to be introspected and connected.

Data types

Let us review the classes of data i/o undertaken a digital twin:

Config

Configuration data (input)

Control parameters relating to what the twin should do, or how it should operate. For example, should a twin 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?

These values should always have defaults.

Values

Value data (input, output)

Raw values passed directly to/from a twin. For example current rotor speed, or forecast wind direction.

Values might be passed at instantiation of a twin (typical application-like process) or via a socket.

These values should never have defaults.

Files

File data (input, output)

Twins frequently operate on file content - eg files on disc or objects in a cloud data store. For example, groups of `.csv` files can contain data to train a machine learning algorithm. There are four subclasses of file i/o that may be undertaken by digital twins:

1. Input file (read) - eg to read input data from a csv file
2. Temporary file (read-write, disposable) - eg to save intermediate results to disk, reducing memory use
3. Cache file (read-write, persistent) - eg to save a trained classifier for later use in prediction
4. Output file (write) - eg to write postprocessed csv data ready for the next twin, or save generated images etc.

External

External service data (input, output)

A digital twin might:

- GET/POST data from/to an external API,
- query/update a database.

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 schema-compliant.

Credentials

Credentials (input)

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.

Credentials should never be hard-coded into application code, always passed in

Monitors/Logs

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.

Data descriptions

Here, we describe how each of these data classes is described by **twined**.

Config

Configuration data

Configuration data is supplied as a simple object, which of course can be nested (although we don't encourage deep nesting). The following is a totally hypothetical configuration...

```
{
    "max_iterations": 0,
    "compute_vectors": True,
    "cache_mode": "extended",
    "initial_conditions": {
        "intensity": 0.0,
        "direction", 0.0
    }
}
```

Values

Value data (input, output)

For Values data, a twin will accept and/or respond with raw JSON (this could originate over a socket, be read from a file or API depending exactly on the twin) containing variables of importance:

```
{
    "rotor_speed": 13.2,
    "wind_direction": 179.4
}
```

Files

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": [
        {
```

(continues on next page)

(continued from previous page)

```
            "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",
                    "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 ...
        }
    ]
}
```

Note:

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

"Uh, so I need an ordered sequence of files, that are CSV files, and are tagged as lidar."

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.

External

External service data (input, output)

There's nothing for **twined** to do here!

If the purpose of the twin (and this is a common scenario!) is simply to fetch data from some service then return it as values from the twin, that's perfect. But its the twin developer's job to do the fetchin', not ours ;)

However, fetching from your API or database might require some credentials. See the following tab for help with that.

Credentials

Credentials (input)

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
```

twined helps by providing a small shim to check for their presence and bring these environment variables into your configuration.

> **Attention:** 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.

Monitors/Logs

Monitor data (output)

Log data (output)

> **Attention:** *What's the difference between Configuration and Values data? Isn't it the same?*
>
> No. Configuration data is supplied to a twin to initialise it, and always has defaults. Values data is ingested by a twin, maybe at startup but maybe also later (if the twin is working like a live server). In complex cases, which Values are required may also depend on the Configuration of the twin!
>
> Values data can also be returned from a twin whereas configuration data is not.
>
> Don't get hung up on this yet - in simple (most) cases, they are effectively the same. For a twin which is run as a straightforward analysis, both the Configuration and Values are processed at startup.

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 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 digital twin schema* mention human-readability as a must; this paper suggests a 20% slower comprehension of camel case than snake.
- 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.
- 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.

4.6 License

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

4.6.1 The boring bit

4.6.2 Third Party Libraries

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

4.7 Version History

4.7.1 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.

4.7.2 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

Backward Incompatible API Changes

1. n/a (Initial release)

Bug Fixes & Minor Changes

1. n/a (Initial Release)