expipe Documentation

Release 0.5.1

Svenn-Arne Dragly, Milad H. Mobarhan, Mikkel E. Lepperød

Jun 25, 2021

Contents

1 Contents **3**
1.1 Introduction . . . 3
1.2 Installation . . . 4
1.3 Getting started . . . 5
1.4 Making a plugin . . . 7
1.5 Developers' guide . . . 8
1.6 Authors . . . 8

2 Indices and tables **9**

Expipe is a lightweight data management platform that aims to simplify the steps from experiment to data analysis. We acknowledge the need for multiple tools, workflows and formats in experimental pipelines and have made an agnostic platform that focuses on adding value and formalizing existing workflows rather than a one-size-fits all solution. Expipe provides functionality to track experimental data and metadata for easy retrieval in exploration and analysis. The goal of Expipe is to simplify data management and allow the user to focus on production and analysis of the data.

CHAPTER 1

Contents

The following sections will guide you through the basics of Expipe for data management of experimental pipelines.

1.1 Introduction

Technological advances are revolutionizing methods in neuroscience. The vast collection of recording setups used, and the large variety of experimental subjects puts high demands on flexible data organization.

Often in neuroscience, the experimental setup is not finalized or rigidly predefined before data acquisition begins. Results may require additional branches of experimentation or reevaluation of the setup. Put simply, experiments have a tendency to organically grow along the experimental time line.

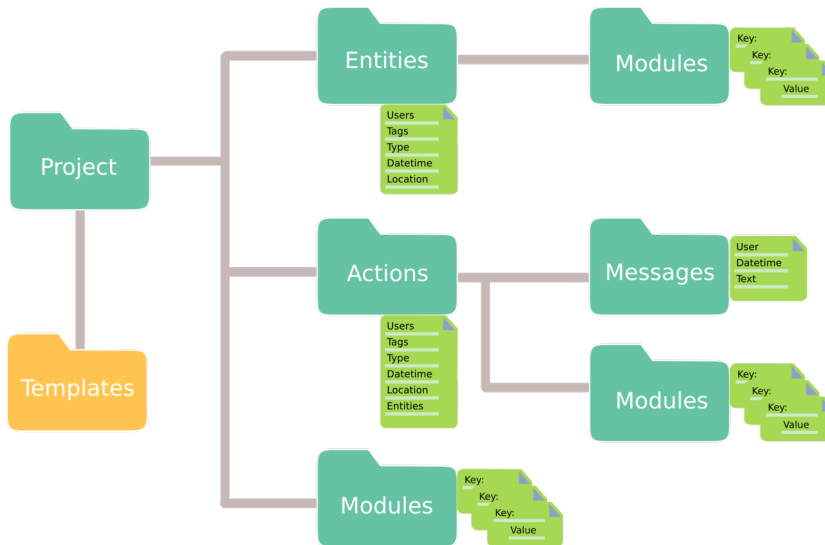
Expipe is thus introduced as an organizational tool that grow organically together with the experimentation - to ease data management in such experimental paradigms.

The aims of Expipe is to organize data and metadata in a way such that they:

- are flexible towards a multitude of user aspects.
- are readable for humans and machines for many years to come.
- are sharable.
- support high throughput data analysis.
- support multiple types of large-scale data sets.

To this end we use the flexible filesystem as a non structured (NoSQL) type database to store data and metadata. This way of storing metadata consist of assigning key-value pairs as Python dictionaries.

An Expipe `project` contains the object types `actions`, `entities`, `modules` and `templates`.



Together, these objects represent an experimental project at large. `modules` sit at the core of the system and are used to describe `projects`, `actions` and `entities` in detail. The `modules` typically contain metadata about the equipment, environment, or subjects, such as the numerical aperture of a microscope lens, the serial number of an acquisition system, or the temperature of a room. `actions` define events that occurred at a specific time, such as an experiment, an analysis, or a simulation. `actions` have a few specific attributes, such as a timestamp, and store detailed metadata in `modules`. Entities are long-lived things that are used in an `action`, typically an experimental subject such as a rat or mouse. `actions` refer to Entities, but but they do not link directly to them. `messages` are user specific lines of text added to `actions`, such as notes.

During an experiment metadata can be automatically added by user specific `templates`. Templates are prefilled key-value pairs describing all aspects of your experiments e.g. recording environment, acquisition system etc. When added, `templates` are introduced as *modules* which are descriptors of `project` and/or `action` entities. A `project` is the root object during communication with `expipe` and contain `modules` and `actions`. `actions` are individual, well actions, of interaction with experimental assets or `expipe` itself such as recordings or analysis respectively. `actions` also contain `modules` which are specific to a particular `action` in contrast to `project modules` which are more general.

We encourage users from neuroscience to base templates on the odML terminologies which can be *a priori* filled out by a user or added in an empty state for *a posteriori* documentation.

1.2 Installation

You can install `expipe` using pip:

```
>>> pip install expipe
```

Alternatively, expipe can be installed from source as follows:

```
>>> git clone https://github.com/CINPLA/expipe.git
>>> cd expipe
>>> python setup.py develop
```

1.3 Getting started

Expipe can be used to manage multiple projects. Each project consists of collections of actions, entities, templates and project modules.

1.3.1 Project

Create a new project with the `require_project` function:

```
import expipe
project = expipe.require_project("test")
```

The default backend for Expipe uses the filesystem. In this case, the above command will create a folder named `test` in your current working directory. Other backends will create a backend-specific project in its database.

1.3.2 Actions

Actions are events that are performed during a project. An action can be an experiment, preparations for an experiment, or an analysys performed after an experiment.

To create an action on the project or return an existing action if it already exists, use `project.require_action`:

```
action = project.require_action("something")
```

1.3.3 Action attributes

To give actions easily searchable properties you can add `Tags`, `Users`, `Subjects` and `Datetime`

```
from datetime import datetime
action.tags = ['place cell', 'familiar environment']
action.datetime = datetime.now()
action.location = 'here'
action.type = 'Recording'
action.subjects = ['rat1']
action.users = ['Peter', 'Mary']
```

1.3.4 Modules

Actions have multiple properties such as the type, location, users, tags and subjects. If you want to expand an action with more information, you can use modules. Modules can hold arbitrary information about the action and can be predefined by using templates to make it easy to add the same information to multiple actions. Ideally, templates should be designed in the beginning of a project to define what should be registered in each action.

To add a module to an action, use `require_module`. The function takes an optional `template` parameter:

```
tracking = action.require_module("tracking", template="tracking")
```

If you are not using templates you may also create modules using dictionaries:

```
import quantities as pq
tracking_contents = {'box_shape': {'value': 'square'}}
tracking_module = action.require_module(name="tracking",
                                        contents=tracking_contents)
elphys_contents = {'depth': 2 * pq.um, }
elphys_module = action.require_module(name="electrophysiology",
                                      contents=elphys_contents)
```

You can loop through modules in an action similarly to a dictionary:

```
for name, val in action.modules.items():
     if name == 'electrophysiology':
         print(val['depth'])
```

```
2.0 um
```

To further retrieve and edit the values of a module, you can use the `module.to_dict()`:

```
tracking = action.require_module(name="tracking")
print(tracking.to_dict())
```

```
OrderedDict([('box_shape', {'value': 'square'})])
```

1.3.5 From Template to Module

To upload a template you can write it as a `dict` and use `require_template`.

```
daq_contents = {
    "channel_count": {"definition": "The number of input channels of the DAQ-device.",
                      "value": "64"}}
expipe.require_template(template='hardware_daq',
                        contents=daq_contents)
```

In order to use a template and add it as a module to an `action` use `action.require_module`:

```
daq = action.require_module(template='hardware_daq')
```

Now, the template `hardware_daq` is added to your action as a module and you also have it locally stored in the variable `daq`. To retrieve `daq` keys and values use `to_dict`:

```
daq_dict = daq.to_dict()
print(daq_dict.keys())
```

```
odict_keys(['channel_count'])
```

```
print(daq_dict.values())
```

```
odict_values([{'definition': 'The number of input channels of the DAQ-device.', 'value
↪': '64'}])
```

1.3.6 Messages

If you want to expand an action with notes and messages, you can use `messages`. Messages are annotations from users that are involved with an action. To add a message to an `action` you can run:

```
from datetime import datetime
messages = [{'message': 'hello', 'user': 'Peter', 'datetime': datetime.now()}]
action.messages = messages
```

1.4 Making a plugin

This section describes how to make a plugin for the Expipe command line interface (CLI). For the complete example; see https://github.com/CINPLA/expipe-plugin-example.

In order to make a plugin for the comman line interface you first need to make a python package.

Begin by making a folder named `my_plugin` with a module, let's call it `my_module.py` containing:

```
from expipe.cliutils import IPlugin
import click


class MyPlugin(IPlugin):
    """Create the `expipe print-me-stuff` command."""
    def attach_to_cli(self, cli):
        @cli.command('print-me-stuff')
        @click.argument('stuff', type=click.STRING)
        def print_me_stuff(stuff):
            '''
            Print stuff

            COMMAND: stuff
            '''

            print(f'Expipe is printing: {stuff}')
```

The folder `my_plugin` must also contain a file `__init__.py` containing:

```
from .my_module import MyPlugin
```

In the root directory you need a `setup.py` file with the following minimum content:

```
from setuptools import setup

from setuptools import setup, find_packages

setup(
    name="my_plugin",
    packages=find_packages(),
    include_package_data=True,
)
```

After the plugin package is ready, all you need to do is to install it and add it to the Expipe environment:

```
>>> python setup.py develop
>>> expipe config global --add plugin my_plugin
```

Finally, you can run your new incredible plugin with expipe:

```
>>> expipe print-me-stuff "Hey! This is my first Expipe plugin!"
```

```
Expipe is printing: Hey! This is my first Expipe plugin!
```

1.5 Developers' guide

1.5.1 Getting the source code

We use the Git version control system. The best way to contribute is through GitHub. You will first need a GitHub account, and you should then fork the repository.

1.5.2 Working on the documentation

The documentation is written in reStructuredText, using the Sphinx documentation system. To build the documentation:

```
>>> cd expipe/docs
>>> make html
```

Then open *doc/build/html/index.html* in your browser.

1.5.3 Committing your changes

Once you are happy with your changes, **run the test suite again to check that you have not introduced any new bugs**. Then you can commit them to your local repository:

```
>>> git commit -m 'informative commit message'
```

If this is your first commit to the project, please add your name and affiliation/employer to `doc/source/authors.rst`

You can then push your changes to your online repository on GitHub:

```
>>> git push
```

Once you think your changes are ready to be included in the main Expipe repository, open a pull request on GitHub (see https://help.github.com/articles/using-pull-requests).

1.6 Authors

Expipe is developed and maintained by CINPLA (Center for Integrative Neuroplasticity), at the University of Oslo.

Here is a list of the main contributors to the code:

- Mikkel Elle Lepperød
- Svenn-Arne Dragly
- Milad Mobarhan
- Alessio Buccino

CHAPTER 2

Indices and tables

- genindex
- modindex
- search