Orchd SDK

Release 0.1

Mathias Santos de Brito

Jul 20, 2023

CONTENTS:

Welcome to the Orchd SDK documentation page. Here you will find all the necessary information to develop your extensions to Orchd. The *orchd-sdk CLI.* will help you with boilerplate code generation and project management making it easy to extend orchd to your needs. To better understand how orchd and the orchd-sdk are related, check the Orchd Architecture section of this documentation.

CHAPTER

ONE

ABOUT ORCHD SDK PROJECT

1.1 Orchd Architecture

Note: A more detailed view of the Orchd's architecture can be found here: Orchd Documentation: Architecture

The Orchd solution was architected to work as an integrator for Edge and IoT environments. Its two base premises are: "standardization takes time", "without standardization integration is necessary". Orchd stands for "Orchestration Daemon" its intention is to integrate and orchestrate services that barely or cannot communicate with each other.

Orchd development is centered on its architecture, and invites developers to add its own extensions and set the interaction between different systems. With this in mind, *Orchd* offers an SDK to support developers on the process of extending *Orchd*.

In this documentation you will learn how to use the *orchd-sdk* CLI command to create your extensions.

1.2 License

CHAPTER

TWO

ORCHD SDK CLI USAGE

The Orchd SDK CLI is a tool that make it easy extend the Orchd. It generates boilerplate code for Sensors, Sinks and Reactions so that you can start coding your Application Logic right away.

Contents

2.1 Installing Orchd-SDK

The *orchd-sdk* CLI is installed when you install the *orchd-sdk* with pip. To install *orchd-sdk* just run:

```
$ pip install orchd-sdk
```

2.2 Orchd Extension Project

A extension project in Orchd is a collection of custom sensors, reactions, sinks and communicators. To build them you have to derive your own classes from the base classes offered by Orchd SDK, they are:

- AbstractSensor
- AbstractReaction
- AbstractSink

After implementing them you have to pack them and distribute, e.g. via PyPi. To make this process easier the *Orchd SDK* offers an CLI that can be invoked by using *orchd-sdk* in your shell after you install *Orchd SDK*.

orchd-sdk command you allow you to create a project to hold your new Orchd Extensions, it will generate the boilerplate code for your new Senors, Reactions and Sinks as well manage the build and deployment of your code. The workflow is simple:

- Create a project with *orchd-sdk new project*
- Enter in project directory and review the files generated.
- Start creating your extensions with *orchd-sdk new [sensor | reaction | sink]*
- Create your tests (we encourage you to write testes)
- **Add your code to the Sensor/Reaction/Sink module created.**
  - Inside the main package you will find the *sensors*, *reactions* and *sinks* subpackages where the respective extension is added as a module (per sensor/reaction/sink). So, if you create two sensors, there will be two modules inside the *sensors* subpackage, the same happens to sinks and reactions.
- Run your tests and certify that everything works.
- **Deploy to PyPi with *orchd-sdk deploy***
  - The *deploy* command will allow you to try your package in the PyPi test server, we recommend you to do so.
  - Note that if you upload a version, you will not be able to override it, to deploy again you need to provide a new version to your project. Also you can use build numbers in your project, so if you upload more than one package with the same version but with different build tag, PyPi will distribute the package with the highest build number, e.g. *my_package-0.1-1* where the last 1 is the build version.
- Now, in your Orchd environment you can install the packages and start using them right away without, even, the need to restart the *orchd daemon*, just go adding the sensors and reactions to it by using the *orchd* CLI.

2.2.1 Creating a new Project

Now that you understands what an *orchd-sdk* project is, let's take a look on how to create one and how it is structured. To create a new project type the following in your terminal:

```
$ orchd-sdk new project
```

It will ask you some questions and configure the project accordingly. You will have to provide the following information:

- The project name using kebab case (words separated by dash).
- The name of the author of the extensions.
- The initial version of the project.
- A description of the project.

If you prefer to pass these values in a non-iterative manner you can type *orchd-sdk new project –help*, the following options are available:

```
$ orchd-sdk new project --help

  Usage: orchd-sdk new project [OPTIONS]

Create a new Project

Options:
  -n, --name TEXT         Name of the project
  -ns, --namespace TEXT   Project namespace
  -a, --author TEXT       Project author
  -l, --license TEXT      Project license
  -v, --version TEXT      Version of the Project
  -d, --description TEXT  Project description
  --help                  Show this message and exit.
```

2.2.2 Project Structure

After you create a project some files and folder will be created and the structure of your project will be the following:

```
├── README.md
├── VERSION
├── orchd.meta.json
├── requirements.txt
├── setup.cfg
├── setup.py
├── src
│   └── my_project_name
│       ├── __init__.py
│       ├── reactions
│       │   └── __init__.py
│       ├── sensors
│       │   └── __init__.py
│       └── sinks
│           └── __init__.py
└── tests
    ├── test_reactions.py
    ├── test_sensors.py
    └── test_sinks.py
```

- *README.md*, use this file to add basic documentation and infos about your project.
- *VERSION*, file holding the current version of your project
- *orchd.meta.json*, file holding metadata about the project, e.g. those you type when you create a new project.
- *requirements.txt*, python project file containing your python dependencies.
- *setup.cfg* and *setup.py* are the files with the information necessary to build your extension, you can always edit them if necessary.
- *src* folder holds your project source folder
- *src/my_project_name* your project's main source package.

- *src/my_project_name/reactions* subpackage holding reactions' modules.
- *src/my_project_name/sensors* subpackage holding sensors' modules.
- *src/my_project_name/sinks* subpackage holding sinks' modules.

2.3 Creating Extensions - Reactions, Sensors and Sinks

After the initialization of your project, you can start creating extensions: reactions, sensors and sinks. You can use for that purpose the subcommand *new* of *orchd-sdk* CLI followed by the kind of extension you want to create, the available options are *reaction*, *sensor* and *sink*.

Warning: Always run the subcommand *new* of *orchd-sdk* in the root directory of your project.

2.3.1 Creating a new Reaction

Note: To know more about *Orchd Reactions* follow this link: Introduction to Orchd Reactions

To create a new reaction run:

```
$ orchd-sdk new reaction
```

You will need to answer some questions regarding your new reactions, they are:

- The reaction module name in snake case, e.g. container_start
- The reaction initial version, this is important because you my need to version your reactions.
- The list of events' names that triggers this reaction in the form of a python list, e.g. *["io.orchd.events.system.Test", "com.example.events.SomeEvent2"]*
- The handler parameters, they are default input attributes that are going to be available to the reaction, it can be for example and endpoint to some service. It takes the form of a JSON string. These values can be changed when deploying your reaction on *orchd*.

You can also provide the values with a non-interactive approach, to check the options just use *–help*, the options are listed below:

```
Usage: orchd-sdk new reaction [OPTIONS]

  Creates a new Reaction

Options:
  -n, --name TEXT
  -v, --version TEXT
  -t, --triggers TEXT
  -hp, --handler_params TEXT
  -a, --active BOOLEAN
  --help                      Show this message and exit.
```

Let's say that you created a new reaction called *container_start*, after you create it you can edit the module source code that is placed in the folder *src/my_project_name/reactions/container_start.py*. The code will look like the one below:

```
import json
import logging

import pydantic

from orchd_sdk.errors import ReactionError
from orchd_sdk.models import ReactionTemplate, Event
from orchd_sdk.reaction import Reaction, ReactionHandler

REACTION_NAME = 'com.mynamespace.ContainerStartReaction'

logger = logging.getLogger(__name__)


class ContainerStartReaction (Reaction):

    template = ReactionTemplate(
        name='com.mynamespace.ContainerStartReaction',
        version='0.1',
        triggered_on=["io.orchd.events.system.Test"],
        handler='my_custom_orchd.reactions.container_start.ContainerStartReactionHandler
↪',
        handler_parameters={},
        sinks=[],
        active=True
    )

    def __init__(self, reaction_template: ReactionTemplate):
        super.__init__(ContainerStartReaction.template)


class ContainerStartReactionHandler (ReactionHandler):

    def handle(self, event: Event, reaction: ReactionTemplate):
        """Implement here the logic of you reaction handler"""
        logger.info(f"com.mynamespace..ContainerStartReaction: Event {event.id} Captured␣
↪and Handled...")
```

Note: You can edit the template class attribute as you wish, it will be used as a base to create templates for this *reaction* when using *orchd*.

Now you have to implement your logic, *orchd* will execute the *handle* method of the ReactionHandler, the setup and initialization of the reaction is done by *orchd*, in most of the cases you just need to implement the *handle* method.

The generated reaction works right away if you leave the default values, so you can deploy on *orchd* and test it, but of course you want to add your own logic. This is very useful if you are in a development environment and you used the *pip install -e .* command, that updates the source of your package as soon as you save it, all you need to try it with *orchd* is to restart the daemon (ps. your package and the *orchd* daemon must be using the same python environment/virtual environment).

2.3.2 Creating a new Sink

Note: To know more about *Orchd Sensors* follow this link: Introduction to Orchd Sinks

Sinks are elements associated to Reactions, it is used when you want to "deliver" the data to other system like a database or a Rest API. You can associate as many sensors as you want to an reaction, an this is a very nice architectural feature of *orchd*.

You may want to write your own sinks to deliver data to support systems, legacy systems, to the cloud, etc. To create a new *sink* run the following command:

```
$ orchd-sdk new sink
```

You will have to answer some questions, they are:

- The name of the sink in snake case (words separated by underscore _)
- The version of the sink, you may want to provide independent versioning to skins.
- Sink default parameters as a JSON string, these values can be changed when deploying the sink along with a reaction.

You can also provide the values you answer in a non-interactive way, use *–help* to get the available options:

```
$ orchd-sdk new sink --help
Usage: orchd-sdk new sink [OPTIONS]

  Creates a new sensor

Options:
  -n, --name TEXT        Name of the Sink module. (snake case)
  -v, --version TEXT     Sink version
  -p, --parameters TEXT  sink parameters
  --help                 Show this message and exit.
```

After you finish the process a new module with the name given will be created under your *sinks* subpackage of your project. Suppose you create a sink called *aws_s3*, the module source file content will look like this:

```
import logging

from orchd_sdk.models import SinkTemplate
from orchd_sdk.sink import AbstractSink

logger = logging.getLogger()


class AwsS3Sink (AbstractSink):
    """Dummy Sink for testing purposes"""

    template = SinkTemplate(sink_class='my_custom_orchd.sinks.aws_s3.AwsS3Sink',
                            name='com.mathiasbrito.aws_s3.AwsS3Sink',
                            version='0.1',
                            properties={})

    def __init__(self, template: SinkTemplate):
```

(continues on next page)

(continued from previous page)

```
        super().__init__(template)

    async def sink(self, data):
        logger.info(f'{AwsS3Sink}: Data SUNK! Actually, I did Nothing! :P {data}')

    async def close(self):
        pass
```

Adjust the *docstrings* and implement your logic on *async def sink(self, data)* method. This codes is ready to be deployed but obviously you will want to add your code. Also you can change the template class attribute this is used as the basis to construct Sink Templates, where you can provide specific properties, for example the AWS S3 endpoint.

2.3.3 Creating a new Sensor

Note: To know more about *Orchd Sensors* follow this link: Introduction to Orchd Sensors

Sensors in Orchd are software entities responsible for capturing information and inject it on Orchd Reactor in the form of an event. The reactor, based on the event, triggers the appropriate reactions. *orchd-sdk* offers a command to create the sensor's boilerplate code. The code is ready to be deployed, but obviously you want to customize the code.

To create a new Sensor run the following command:

```
$ orchd-sdk new sensor
```

The command will ask you some information about the new sensor being created, they are:

- The sensor module name in snake case.
- A sensor's brief description.
- The sensor version, you may want to track versions in a different pace from your main package.
- Sensor's default parameters to be used when starting a sensor instance. JSON format as string.
- The default sensing interval, how often the sensor must pull information from the environment.

Note: The sensing interval must be used if you are using an PULL strategy, let's say you want to capture the temperature from a physical sensor each 10 seconds. If you want to use a PUSH approach you may reduce this variable to 0, but be sure that you use *async/await* to avoid blocking, which can interfere in the performance of your sensor and the system.

You can provide the values for the questions in a non-interactive way by providing the related parameters to the command, they are:

```
Usage: orchd-sdk new sensor [OPTIONS]

  Creates a new sensor.

Options:
  -n, --name TEXT                 Sensor module name (use snake case)
  -d, --description TEXT          Brief description for the new sensor.
  -v, --version TEXT              Version number for the sensor.  [default:
                                  0.0]
```

(continues on next page)

(continued from previous page)

```
  -sp, --sensor-param TEXT        Sensor Parameters as JSON
  -si, --sensing-interval INTEGER
                                  Sensing Interval in seconds (int)  [default:
                                  1]
  -c, --communicator TEXT         [default:
                                  orchd_sdk.sensor.LocalCommunicator]
  --help                          Show this message and exit.
```

As you can see, you can change the communicator, by default this version of *orchd-sdk* will automatically use the LocalCommunicator, the only provided by *Orchd* right now, new types of communicators are constantly being develop to provide different ways to communicate with the Orchd Reactor, for more information you may want to check the Orchd Architecture.

You will find the code for your new sensor in a new python module, with the name you provided, inside the *sensors* folder of your project. Let's say that I want to sense the event of a container being stopped, and I gave my sensor the name *container_stopped*, the initial code would look like this:

```python
import asyncio
import logging

from orchd_sdk.models import SensorTemplate, Event
from orchd_sdk.sensor import AbstractSensor, AbstractCommunicator, SensorState


logger = logging.getLogger(__name__)


class ContainerStoppedSensor (AbstractSensor):
    """
    Sensor that emits XYZ events.
    """
    template = SensorTemplate(
        name='com.mathiasbrito.ContainerStoppedSensor',
        description='This sensor listend to container stop events',
        version='0.1',
        sensor='my_custom_orchd.sensors.container_stopped.ContainerStoppedSensor',
        communicator='orchd_sdk.sensor.LocalCommunicator',
        parameters={},
        sensing_interval=1
    )

    def __init__(self, sensor_template,  communicator):
        super().__init__(sensor_template, communicator or 'orchd_sdk.sensor.
↪LocalCommunicator')
        self.state = SensorState.READY


    async def sense(self):
        logger.info(f'{self.template.sensor}')
        await self.communicator.emit_event(
            Event(event_name='io.orchd.events.system.Test', data={'dummy': 'data'})
        )
```

Warning: Note that the sense method is *async* and by using the *LocalCommunicator* you must ensure that your code is asynchronous and you not block. Implement you sensing code asynchronous and wait for it inside *sense*.

After changing the sensor's code, creating the tests, and making sure that your tests pass, your sensor must be ready to be build and deployed.

2.3.4 Testing your Project

The Orchd Extension Project structure provides you with a tests directory with some files you can use to write tests for your sensors, reactions and sinks. Orchd extension projects uses *pytests* and is pre-configured to run the tests. Just give it a try and run:

```
$ orchd-sdk test
```

Other commands will invoke test before starting, this is the case for *orchd-sdk build* and *orchd-sdk deploy*.

2.3.5 Building

You can use *orchd-sdk* to invoke the python build process, it is a simple wrapper around *setuptools*. Since in the future the *orchd-sdk* can require additional steps, it was decided to add a level of abstraction in the building process so that we can accommodate future changes in the building process without compromising the user development workflow.

To build your project package run:

```
$ orchd-sdk build
```

orchd-sdk will run the tests and if them pass it will build the packages, they will be created in the *build* directory. If, for some reason, you want to skip the tests (BE CAREFUL) you can provide the *–skip-tests* command as *$ orchd-sdk build –skip-tests*.

2.3.6 Deploying

You may want to deploy your new Orchd Extension to PyPi so that it can be easily installed with *pip*. To do so you can use the *deploy* command, like this:

```
$ orchd-sdk deploy
```

By default the deploy command will run *test* and *build* commands for you and if everything goes well it will try to deploy it to the PyPi Test Server.

Note: We decide to set *orchd-sdk deploy* behavior to not send the package to PyPi main server right away, to do so you have to provide the *–upload* option. The main reason is to make sure that you will review your work before, one more time before deploying.

If the tests and build runs ok, the command will start the deployment process to the PyPi test server, make sure to create and account and provide the credentials when asked to (*orchd-sdk* uses twine to deploy your package). You can check the result by visiting the PyPi test server, usually your project url will be *https://test.pypi.org/project/my-project-name/*.

If you fell comfortable with the results, do the final deployment by adding the *–upload* to the command, provide your credentials and you are done with the first version of your Orchd Extension. After a successful deployment you can use

pip to install your extension and start deploying sensors, reactions and sinks using the *orchd* CLI, for mor information see the Orchd Documentation.

2.3.7 Creating Templates

When you develop a new Sensor/Reaction/Sink you set a *template* class attribute, this attribute is used as a base to create new custom templates for Sensors/Reaction/Sinks based on your class. Templates tells *orchd* how to setup and deploy the new Sensor/Reaction/Sinks, one attribute that probably you want to change are the *name*, *description* and *parameters*, for example, let's say you developed a sensor that detects when a container is stopped, but you are interested only in those events related to containers tagged with some value, you can implement your sensor in a way that it will use the tag of interest from a *parameter*.

The idea behind the templates is that you can have multiple templates based on your custom Sensor/Reaction class so that you can model different scenarios and different deployment environments. *orchd-sdk* provides you with a command to generate a base template file out of a *orchd* extension class' default template class attribute value.

Warning: To use *orchd-sdk template* to create a template file out of your class, make sure that your package containing the class is installed.

Run the command:

```
$ orchd-sdk template --from my_package.sensors.MySensor
```

The command will print a JSON in your terminal containing the default values for your extension class, for example, if we run the command for the *orchd* test DummySensor, we get the following:

```
$ orchd-sdk template --from orchd_sdk.sensor.DummySensor

{
  "id": "71fa0f62-22e0-4068-b5c6-ba9f46b739ac",
  "name": "io.orchd.sensor_template.DummySensor",
  "version": "1.0",
  "sensor": "orchd_sdk.sensor.DummySensor",
  "sensing_interval": 0.0,
  "communicator": "orchd_sdk.sensor.LocalCommunicator",
  "parameters": {
    "some": "data"
  },
  "description": "A dummy Sensor to be used for testing purposes"
}
```

You can redirect the output to a file and edit it. After that you can add this template to Orchd and start deploying extensions based on these templates. To understand how to deploy sensors/reactions/sinks see the orchd documentation.

CHAPTER

THREE

SOURCE CODE REFERENCE

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search