Data Engineering with Dagster — Week 3

Testing

CODE LOCATION

`week_1/tests/test_answer.py`

Testing Our Project

We have actually been using tests since week one. When you ran `pytest` to validate the first two projects, you were running unit tests on your Dagster components. While you will not have to write any tests for this course, it is worth diving a little deeper into how to write effective tests, as well as how testing works with the different abstraction layers.

Simple Tests

Let's revisit our assignment from week one. We wrote the `process_data()` op, which did some minimal processing. We need to provide a list of our custom type `Stocks` to find the correct `Aggregation`. An op this simple actually behaves like any other function. We simply have to provide an input and check the expected output.

```
@pytest.fixture
def stocks():
    return [
        Stock(date=datetime.datetime(2022, 1, 1, 0, 0), close=10.0,
volume=10, open=10.0, high=10.0, low=10.0),
```

```
        Stock(date=datetime.datetime(2022, 1, 2, 0, 0), close=10.0,
volume=10, open=11.0, high=10.0, low=10.0),
        Stock(date=datetime.datetime(2022, 1, 3, 0, 0), close=10.0,
volume=10, open=10.0, high=12.0, low=10.0),
        Stock(date=datetime.datetime(2022, 1, 4, 0, 0), close=10.0,
volume=10, open=10.0, high=11.0, low=10.0),
    ]

def test_process_data(stocks):
    assert process_data(stocks) == Aggregation(date=datetime.datetime(2022,
1, 3, 0, 0), high=12.0)
```

What about the `get_s3_data` op? That required a configuration setting applied at the job level. Luckily, Dagster offers `build_op_context` which makes setting op configurations simple.

```
@pytest.fixture
def file_path():
    return "week_1/data/stock.csv"


def test_get_s3_data(file_path):
    with build_op_context(op_config={"s3_key": file_path}) as context:
        get_s3_data(context)
```

Finally, to test an entire job, we just have to set the necessary configurations at the job level. For week one, we can test whether the entire job functions correctly:

```
def test_job(file_path):
    week_1_pipeline.execute_in_process(run_config={"ops":
{"get_s3_data": {"config": {"s3_key": file_path}}}})
```

Tests with Resources

Writing tests that rely on external resources is the hard part of data engineering tests. Testing can become a chore in mocking, with unit test code becoming so complicated that you lose track of what you are mocking.

However, we know from last week that working with external resources is one of Dagster's best attributes. It is easy to plug in different definitions of resources depending on the situation. To apply this to unit tests, we just need to sub in mocks for resources.

In fact, we already did this! When we interacted with S3 and Redis locally in week two, we used `ResourceDefinition.mock_resource()`, which uses a `MagicMock` for our resource. This is perfect for writing unit tests! Let's look at the pipeline test for week two. For our `redis` resource, we are using the `ResourceDefinition.mock_resource()`:

```
def test_week_2_pipeline(stock_list):
    week_2_pipeline.execute_in_process(
        run_config={"ops": {"get_s3_data": {"config": {"s3_key":
"data/stock.csv"}}}},
        resources={"s3": mock_s3_resource, "redis":
ResourceDefinition.mock_resource()},
    )
```

You may see that the `s3` resource actually uses a special mock, `mock_s3_resource`. Sometimes a simple `MagicMock` is not enough. Sometimes we need to set the return values for the calls made by the resource.

The `get_s3_data()` op is a good example. That op relies on a resource for a client where the data must be returned for processing.

```
@pytest.fixture
def stock_list():
    return ["2020/09/01", "10.0", "10", "10.0", "10.0", "10.0"]

def test_get_s3_data(stock_list):
    s3_mock = MagicMock()
    s3_mock.get_data.return_value = [stock_list] * 10
    with build_op_context(op_config={"s3_key": "data/stock.csv"}, resources=
{"s3": s3_mock}) as context:
        get_s3_data(context)
        assert s3_mock.get_data.called
```

Here, we are creating a `MagicMock` within the test itself and setting the return value for the `get_data()` method. We can then pass this mock to the `s3` resource within the `build_op_context`. After that setup, we can execute our op.

Summary

Whatever abstraction layer you need to test in Dagster, there is a way to set it within tests. At Dutchie, we ensure all of our Dagster code has 100% test coverage. Outside of Dagster, there are not many data engineering frameworks that realistically allow for full test coverage. As you set up Dagster in production, it can be helpful to review the tests associated with each week to get a sense of how you can apply these to your own project.

LECTURE PREVIEW

With the ability to mock resources, it is worth considering what type of resources you should use with your tests. We have already seen that we can run our pipelines against mocks (what we have been calling "local) or against Dockerized versions of resources (such as localstack). Both can run locally, so which should we use?

Tests are a real difference maker with Dagster and we will dive into best practices during the lecture.