Qiskit Backends: what they are and how to work with them

medium.com/qiskit/qiskit-backends-what-they-are-and-how-to-work-with-them-fb66b3bd0463

May 18, 2021

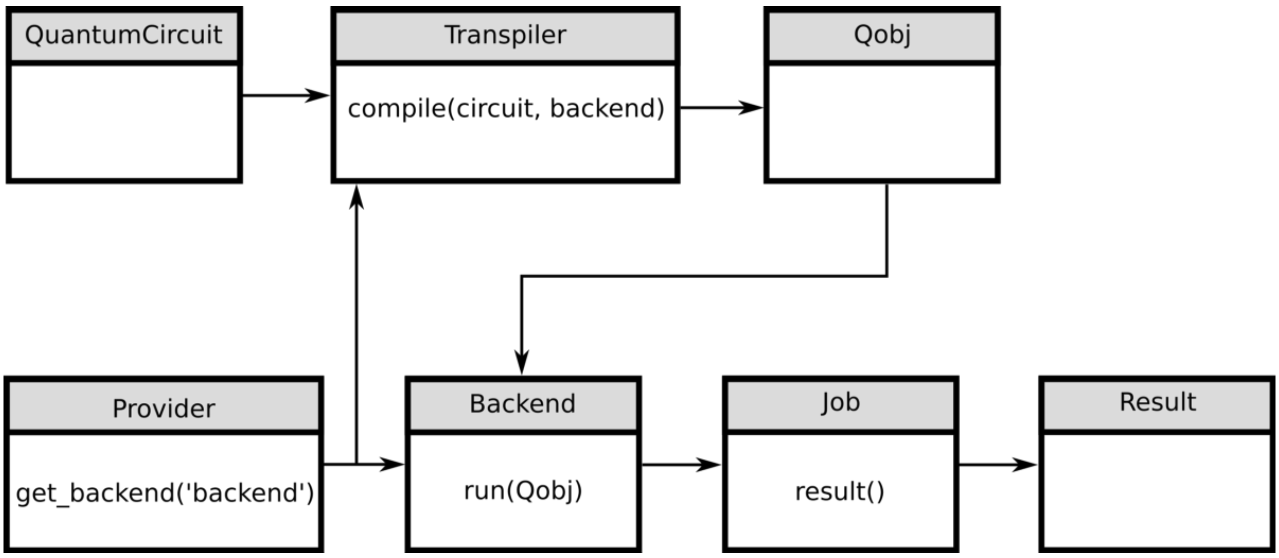
Note: This article is a relic of quantum computing past, and things may not work the same way that they work today.

Erick Winston and Diego Moreda [Technical Writer: Abigail Cross]

In Qiskit 0.6, we have updated the interface for backends and jobs. In this post, we will review the core components of Qiskit's backend framework, using examples to guide you through its advanced features.

The new interface has three parts: the `provider`, the `backend`, and the `job`:

- `provider`: accesses backends and provides backend objects
- `backend`: runs the quantum circuit
- `job`: keeps track of the submitted job



Qiskit 0.6 workflow

Providers

A Provider is an entity that provides access to a group of different backends (for example, backends available through IBM Q). It interacts with those backends to, for example, find out which ones are available, or retrieve an instance of a particular backend.

A provider inherits from `BaseProvider` and implements the methods:

- `backends()`: returns all backend objects known to the provider.
- `get_backend(name)`: returns the named backend.

Qiskit includes interfaces to two providers: Aer and IBMQ:

- `Aer` : provides access to that are included with Qiskit and run on your local machine.
- `IBMQ` : implements access to cloud-based backends —— hosted on .

For instance, to get the list of backends from local,

will print a list containing the names of the local backends, similar to:

```
qasm_simulatorqasm_simulator_pystatevector_simulatorstatevector_simulator_pyunitary
```

If you already know the name of the backend you want, you can access an instance of it using:

```
backend = Aer.get_backend('qasm_simulator')
```

New in 0.6 are also methods to simplify the handling of credentials for the IBMQ provider. They avoid some of the issues which were seen previously when storing credentials in a `Qconfig.py` file:

- `save_account(token, url=QE_URL, **kwargs)` : Authenticate a new IBMQ account and save credentials to disk for use in a future session.
- `enable_account(token, url=QE_URL, **kwargs)` : Authenticate a new IBMQ account and add for use during this session.
- `active_accounts()` : List all accounts active in current session.
- `load_accounts(**kwargs)` : Load IBMQ accounts found on system into the current session.
- `stored_accounts()` : List all accounts stored to disk.
- `disable_accounts(**kwargs)` : Disable account in current session.
- `delete_accounts(**kwargs)` : Delete saved account from disk.

Backends

Backends represent either a simulator or a real quantum computer, and are responsible for running quantum circuits and returning results. They take in a `qobj` as input, which is a quantum object and the result of the compilation process, and they return a `BaseJob` object. This object allows asynchronous running of jobs for retrieving results from a backend when the job is completed.

At a minimum, backends use the following methods, inherited from `BaseBackend` :

- `run(qobj)` : Submits the `qobj` for running and returns a `job` .
- `configuration()` : Returns the backend configuration dictionary, which describes how the backend is configured (such as its topology or native gate set).
- `properties()` : A dictionary of backend specific properties.
- `name()` : The name of the backend.

- `status()` : A dictionary of the backend's status, such as how many jobs it has queued.

Each backend class can also include backend-specific functionality. For example, since the IBM Q backends are accessed through the cloud, they provide the following methods for retrieving job instances between login sessions — for instance, when the queue for the devices is longer than one wants to keep a Python session running:

- `jobs(limit=50, skip=0, status=None, db_filter=None)` : Returns a list of jobs. The list starts at index `skip` and may include up to `limit` jobs.
- `retrieve_job(job_id)` : Returns the job matching `job_id` .

Jobs

Job instances can be thought of as the "ticket" for a submitted job. They find out the execution's state at a given point in time (for example, if the job is queued, running, or has failed) and also allow control over the job. They have the following methods:

- `result()` : This is a blocking call that returns a `Result` object. Depending on implementation, it may wait indefinitely, or give up after a timeout.
- `status()` : Returns a `JobStatus` object.
- `cancel()` : Attempts to cancel the job. Returns `True` if successful.
- `backend()` : The backend instance which created the job.
- `job_id()` : a string the IBM Q API uses to identify the job.

`ibmq` jobs can also report:

- `creation_date()` : The date the job was created.
- `queue_position()` : An estimate of the position of the job on the server queue.
- `error_message()` : A message from the API if an error was encountered with the job.

Simple example

This initial example demonstrates the different components of the module, focused on an Aer backend. After building the circuit, the Aer provider is queried for a backend named "qasm_simulator". The circuit is compiled into a Qobj, which is then submitted to the backend and returns a `Job` . Once the job is completed, the results are obtained and printed.

> Please note that, for conciseness, the following examples assume that the `qobj` variable is initialized in the same way as in the simple example above.

Using jobs from a remote provider asynchronously

This example delves a little deeper into how the asynchronicity of the Jobs can be used. The call where the circuit is sent to the backend (`backend.run()`) is not blocking; the execution of your program continues while the processing of the job is done in the background.

In this snippet, a simple printing of the job status and estimated queue position is performed, but depending on your needs, this asynchronous feature can be used for tailoring the flow of your application, and is suited for scenarios where full control over the jobs is necessary.

Running parallel execution of jobs

This example shows submitting several jobs in parallel and processing results as they become available.

Qiskit’s new backend framework simplifies running complex experiments and provides a straightforward method of expanding to new backends. Try it out and let us know what you think!