SDK Reference

The PLANQK Quantum SDK provides an easy way to develop quantum code that runs on quantum hardware and simulators supported by the PLANQK Platform. It serves as an extension for both the Qiskit SDK and the Amazon Braket SDK. While Qiskit supports a broad range of devices, the Braket SDK is currently supported exclusively for the QuEra Aquila quantum device. This allows you to seamlessly integrate and reuse your existing Qiskit or Braket code, leveraging the power and familiarity of the frameworks you know best. This integration enables you to seamlessly adapt and reuse your existing Qiskit or Braket code within the PLANQK environment, maximizing productivity while working with the frameworks you are already accustomed to.

If you are using PennyLane to implement your quantum machine learning algorithms, you can use the SDK along with the PennyLane-Qiskit plugin to run them on the quantum hardware provided by the PLANQK Platform.

Installation

The package is released on PyPI and can be installed via `pip`:

```bash
pip install --upgrade planqk-quantum
```

Using the SDK

The SDK enables access to quantum hardware and simulators using the Qiskit and the Braket syntax. To access the quantum backends supported by PLANQK via Qiskit you will need to use the `PlanqkQuantumProvider` class. Additionally, you can access the quantum backends provided by PLANQK through AWS Braket using the `PlanqkBraketProvider` class.

Authentication

To use the SDK, you need to authenticate using an access token. You may use your personal access token found on the PLANQK welcome page , or you can generate dedicated access tokens here . This token can be set in two ways:

1. Automatically, by logging in through the PLANQK CLI. The command to login via CLI is `planqk login -t <your_access_token>` . This method will automatically inject the access token when you instantiate the `PlanqkQuantumProvider` class. If you want to log in with your organization you need to additionally execute `planqk set-context` and select the organization.
2. Explicitly, during instantiation of the `PlanqkQuantumProvider` or the `PlanqkQuantumProvider` class, as shown in the example below. This method overrides any access token that has been automatically injected through the PLANQK CLI login. You can optionally pass the organization id as a parameter, if you want to execute your circuit using your organization's account.

If the access token is not set, is invalid, or has expired, an `InvalidAccessTokenError` is thrown. You need to generate a new token and log-in again.

Example Usage

python

```python
from planqk import PlanqkQuantumProvider

# Initizalize the provider in case you are already logged in with the PLANQK CLI
provider = PlanqkQuantumProvider()

# Initialize the provider by passing the access token
provider = PlanqkQuantumProvider(access_token="your_access_token")

# Initialize the provider by passing the access token and organization id
provider = PlanqkQuantumProvider(access_token="your_access_token", organization_id="
```

The `PlanqkBraketProvider` can be instantiated in the same way as the `PlanqkQuantumProvider` . You can optionally specify the `access_token` and the `organization_id` and as a parameter.

```python
from planqk import PlanqkBraketProvider

provider = PlanqkBraketProvider(access_token="your_access_token", organization_id="y
```

Supported Operations

This section provides an overview of the most important classes and methods in the SDK.

PlanqkQuantumProvider

The `PlanqkQuantumProvider` class allows access to all gate-based backends via Qiskit.

Method	Description
`backends()`	This method returns a list of ids of backends supported by PLANQK. Please note that currently, backend filtering is not supported.
`get_backend(backend_id)`	This method returns a single backend that matches the specified ID. If the backend cannot be found, a `PlanqkClientError` is thrown.
`jobs()`	This method retrieves a list of all jobs created by the user, sorted by their creation date with the newest jobs listed first.

If you specify `azure.ionq.simulator` as the backend ID, for example, by calling `provider.get_backend("azure.ionq.simulator")` , a `PlanqkQiskitBackend` is returned.

Qiskit Backends and Jobs

The `PlanqkQiskitBackend` class represents a Qiskit Backend . It provides information about quantum backends (e.g., number of qubits, qubit connectivity, etc.) and enables you to run quantum circuits on the backend. Please note that currently, only circuits with gate-based operations are supported while pulse-based operations are not supported.

The `PlanqkQiskitBackend` class supports the following methods:

Method	Description
`configuration()`	Returns the backend configuration data. This method is included for compatibility with older versions of Qiskit.
`run(circuit, shots)`	Executes a single circuit on the backend as a job (multiple circuits are currently not supported) and returns a `PlanqkQiskitJob` . You also need to specify the number of shots. The minimum and maximum number of supported shots differ for each backend and can be obtained from the backend properties `min_shots` and `max_shots` , respectively. A `PlanqkClientError` is thrown if the job input is invalid or if the designated backend is offline and does not accept new jobs in the moment.
`retrieve_job(job_id)`	Retrieves a job from the backend using the provided id. If a job cannot be found a `PlanqkClientError` is thrown.

This example shows how to run a circuit on a backend:

```python
# Select a certain backend
backend = provider.get_backend("azure.ionq.simulator")

# Create a circuit
circuit = QuantumCircuit(2, 2)
circuit.h(0)
circuit.cx(0, 1)
circuit.measure(range(2), range(2))

# Run the circuit on the backend
job = backend.run(circuit, shots=10)

# Retrieve a job by id
job = backend.retrieve_job("6ac422ad-c854-4af4-b37a-efabb159d92e")
```

Qiskit Jobs & Results

The class `PlanqkQiskitJob` represents a Qiskit Job . It provides status information about a job (e.g., job id, status, etc.) and enables you to access the job result as soon as the job execution has completed successfully.

Methods

Method	Description
`status()`	Returns the status of the job. The Qiskit job states are: `INITIALIZING` , `QUEUED` , `RUNNING` , `CANCELLED` , `DONE` , `ERROR` .
`result()`	Returns the result of the job. It blocks until the job execution has completed successfully. If the job execution has failed, a `PlanqkClientError` is thrown indicating that the job result is not available.
`cancel()`	Cancels the job execution. If the job execution has already completed or if it has failed, this method has no effect.

Results

The type of result depends on the backend where the job was executed. Currently, only measurement result histograms are supported. The histogram is represented as a dictionary where the keys are the measured qubit states and the values are the number of occurrences. The measured qubit states are represented as bit-strings where the qubit farthest to the right is the most significant and has the highest index (little-endian). If supported by the backend, the result also contains the memory of the job execution, i.e., the qubit state of each individual shot.

Attributes

Attribute	Description
`counts`	Returns the histogram of the job result as a JSON dict.
`memory`	Returns the memory as a JSON dict.

Here is an example of how to access these attributes:

```python
result = job.result()
print(result.counts)
# Expected output, e.g., {"11": 6, "00": 4}
print(result.memory)
# Expected output, e.g., ['00', '11', '11', '00', '11', '00', '11', '11', '00', '11'
```

PlanqkBraketProvider

The `PlanqkBraketProvider` class allows access to all backends provided through AWS. This is an overview of the available methods:

Method	Description
`devices()`	This method returns a list of ids of the devices supported by PLANQK through Braket.
`get_device(device_id)`	This method returns a single device that matches the specified ID. If the backend cannot be found, a `PlanqkClientError` is thrown.

If you specify `aws.ionq.forte` as the backend ID, for example, by calling `provider.get_device("aws.ionq.forte")` , a `PlanqkAwsDevice` is returned.

Braket Devices and Tasks

The `PlanqkAwsDevice` class represents an `AwsDevice` and therefore provides the same properties and methods. Below are the key methods and properties:

Property / Method	Description
`status`	Retrieves the current status of the device.
`is_available`	Returns `true` if the device is online and ready to process tasks.
`properties`	Provides the current properties of the device.
`run(task_specification, shots)`	Executes a Braket circuit or an Analog Hamiltonian Simulation program on PLANQK (batch executions are not currently supported) and returns a `PlanqkAwsQuantumTask` . You can specify the number of shots to perform; if not specified, 1000 shots are executed by default. A `PlanqkClientError` is thrown if the task input is invalid or if the device is offline and unable to accept new jobs.

Tasks & Results

The `PlanqkAwsQuantumTask` class is a representation of an AwsQuantumTask . This class provides essential status information about a task, such as its ID, current status, and allows access to its results once the execution is completed successfully.

You can obtain a `PlanqkAwsQuantumTask` object directly from the `run` function of the `PlanqkAwsDevice`. Alternatively, if you need to retrieve a task later, you can create a `PlanqkAwsQuantumTask` object by specifying the task ID. For example, to retrieve a task with the ID `123e4567-e89b-42d3-a456-556642440000`, you would use:

```python
task = PlanqkAwsQuantumTask(task_id="123e4567-e89b-42d3-a456-556642440000")
```

If you are not logged in through the PLANQK CLI, you must also provide your access token, and optionally, your organization ID.

```python
PlanqkAwsQuantumTask(task_id="123e4567-e89b...", access_token="your_access_token", o
```

Methods

Method	Description
`status()`	Returns the current status of the task, which could be `QUEUED`, `RUNNING`, `CANCELLED`, `COMPLETED`, or `FAILED`.
`result()`	Returns the execution result of the task. This method blocks until the task execution completes successfully. If the task fails, a `PlanqkClientError` is thrown, indicating that the result is unavailable.
`cancel()`	Cancels the task execution. If the task has already completed or failed, this method has no effect.

Results

If you execute a Braket circuit the result object is of type `GateModelQuantumTaskResult`. For `AnalogHamiltonianSimulationQuantumTaskResult`. Both result classes include the shot measurements from the execution.

A `GateModelQuantumTaskResult` contains for instance the following properties:

```python
result = task.result()
print(result.measurement_counts)
# Expected output, e.g., Counter({'111': 2, '000': 1})
```

```
print(result.measurements)
# Expected output [[0 0 0][1 1 1][1 1 1]]
```

PennyLane Integration

To use the SDK with PennyLane, you need to install the PennyLane-Qiskit plugin by adding the `pennylane-qiskit` package to your Python project dependencies, e.g., by running `pip install pennylane-qiskit==0.36` .

IMPORTANT

Currently, only `pennylane` and `pennylane-qiskit` packages version 0.36.0 are supported.

To execute a PennyLane circuit using a PLANQK backend, first, retrieve the desired backend using the PlanqkQuantumProvider.
Then, create a `qiskit.remote` device and pass the PLANQK backend to it.

The following example shows how to create a remove device using the `azure.ionq.simulator` backend:

```python
provider = PlanqkQuantumProvider()
backend = provider.get_backend("azure.ionq.simulator")

device = qml.device('qiskit.remote', wires=2, backend=backend, shots=100)

@qml.qnode(device)
def circuit():
    qml.Hadamard(wires=0)
    qml.CNOT(wires=[0, 1])
    return qml.sample(qml.PauliZ(0)), qml.sample(qml.PauliZ(1))

result = circuit()
```

What's next?

- See our supported quantum backends and simulators .

- Check out this Jupyter notebook showing how to utilize the PLANQK Quantum SDK.

Help us improve this page! Last updated: 22.01.25, 23:16

Previous page
Available Backends

Next page
Service Access from Python