

---

# **Qlib-Server Documentation**

**Microsoft**

**Jul 08, 2022**



---

## Contents

---

<b>1</b>	<b>Document Structure</b>	<b>3</b>
1.1	Qlib-Server: Quant Library Data Server	3
1.2	Qlib-Server Deployment	4
1.3	Using Qlib in Online Mode	10
1.4	Changelog	12



Qlib-Server is the assorted server system for Qlib, which utilizes Qlib for basic calculations and provides extensive server system and cache mechanism. With Qlib-Server, the data provided for Qlib can be managed in a centralized manner.

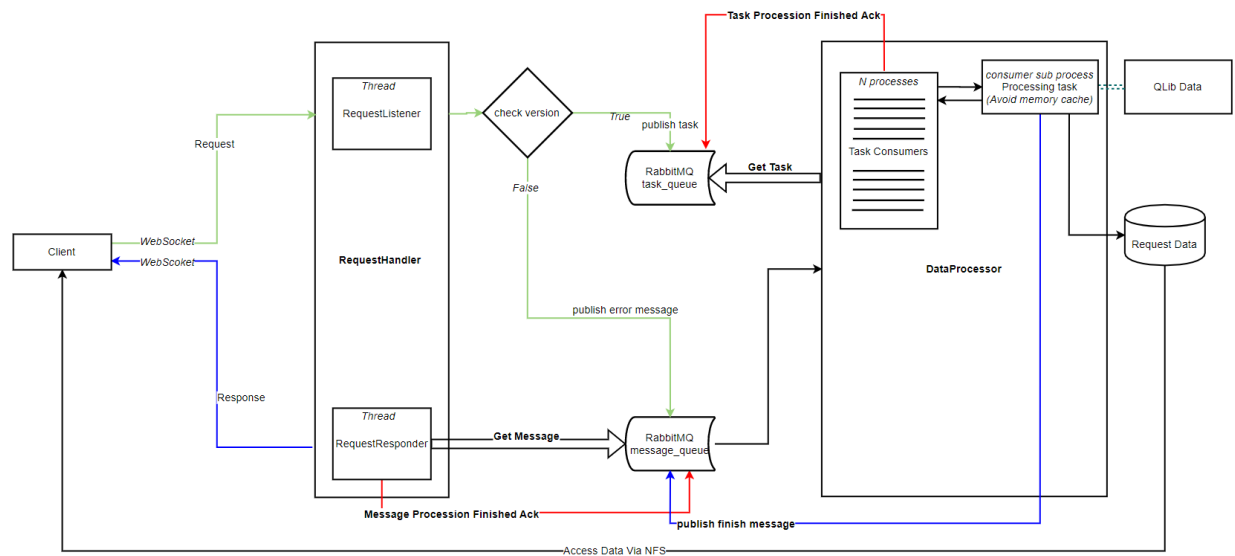


## 1.1 Qlib-Server: Quant Library Data Server

### 1.1.1 Introduction

Qlib-Server is the assorted server system for Qlib, which utilizes Qlib for basic calculations and provides extensive server system and cache mechanism. With Qlib-Server, the data provided for Qlib can be managed in a centralized manner.

### 1.1.2 Framework



The Client/Server framework of Qlib is based on WebSocket considering its capability of **bidirectional communication** between client and server in **async** mode.

Qlib-Server is based on [Flask](#), which is a micro-framework for Python and here [Flask-SocketIO](#) is used for websocket connection.

Qlib-Server provides the following procedures:

### Listening to incoming request from client

The clients will propose several types of requests to server. The server will parse the requests, collect the identical requests from different clients, record their session-ids, and submit these parsed tasks to a pipe. Qlib use [RabbitMQ](#) as this pipe. The tasks will be published to a channel *task\_queue*.

**RequestListener** is used for this function:

After receiving these requests, the server will check whether different clients are asking for the same data. If so, to prevent repeated generation of data or repeated generation of cache files, the server will use [Redis](#) to maintain the session-ids of those clients. These session-ids will be deleted once this task is finished. To avoid IO conflicts, [Redis\\_Lock](#) is imported to make sure no tasks in redis will be read and written at the same time.

### Responding clients with data

The server consumes the result from *message\_queue* and get the session-ids of the clients requiring this result. Then it responds to these clients with results.

**RequestResponder** is used for this method.

The two class above is combined as **RequestHandler**, which is responsible for communicating with clients.

### Accepting tasks from RabbitMQ and processing data

The server will automatically collect tasks from RabbitMQ and process the relevant data. RabbitMQ provides a mechanism that when the server consumes a task, a callback function is triggered. The data processing procedure is implemented within these callbacks and it currently supports the three types of tasks corresponding to above:

- Calendar
- Instruments
- Features

**DataProcessor** is used for this function.

The server will use *qlib.data.Provider* to process the data. RabbitMQ also provides a mechanism that can make sure all tasks is successfully consumed and completed by consumers. This requires the consumer call *ch.basic\_ack(delivery\_tag=method.delivery\_tag)* after successfully processing the data. If the task is not **acked**, it will return to the pipe and wait for another consuming.

Once the task is finished, a result (*could be data or uri*) will be published to another channel *message\_queue*.

## 1.2 Qlib-Server Deployment

### 1.2.1 Introduction

To build a Qlib-Server, user can choose:



- One-click Deployment of Qlib-Server
- Step-by-step Deployment of Qlib-Server

### 1.2.2 One-click Deployment

One-click deployment of Qlib-Server is supported, users can choose either of the following two methods for one-click deployment:

- Deployment with docker-compose
- Deployment in Azure

#### One-click Deployment with docker-compose

Deploy Qlib-Server with docker-compose according to the following processes:

- Install docker, please refer to [Docker Installation](#).
- Install docker-compose, please refer to [Docker-compose Installation](#).
- Run the following command to deploy Qlib-Server:

```
git clone https://github.com/microsoft/qlib-server
cd qlib-server
sudo docker-compose -f docker_support/docker-compose.yaml --env-file_
↪docker_support/docker-compose.env build
sudo docker-compose -f docker_support/docker-compose.yaml --env-file_
↪docker_support/docker-compose.env up -d
# Use the following command to track the log
sudo docker-compose -f docker_support/docker-compose.yaml --env-file_
↪docker_support/docker-compose.env logs -f
```

#### One-click Deployment in Azure

---

**Note:** Users need to have an Azure account to deploy Qlib-Server in Azure.

---

Deploy Qlib-Server in Azure according to the following processes:

- Install azure-cli, please refer to [install-azure-cli](#)
- Add the Azure account to the configuration file `azure_conf.yaml`

```
sub_id: Your Subscription ID
username: azure user name
password: azure password
# The resource group where the VM is located
resource_group: Resource group name
```

- **Execute the deployment script**
  - Run the following command:

```
git clone https://github.com/microsoft/qlib-server
cd qlib-server/scripts
python azure_manager.py create_qlib_cs_vm \
    --qlib_server_name test_server01 \
    --qlib_client_names test_client01 \
    --admin_username test_user \
    --ssh_key_value ~/.ssh/id_rsa.pub \
    --size standard_NV6_Promo \
    --conf_path azure_conf.yaml
```

- To know more about parameters, please run the following command:

```
python azure_manager.py create_qlib_cs_vm -- --help
```

### 1.2.3 Step-by-step Deployment

Users can deploy Qlib-Server step by step, which has the following processes:

- Build RabbitMQ
- Build Redis
- Build NFS
- Build Qlib-Server

#### Build RabbitMQ

RabbitMQ is a general task queue that enables qlib-server to separate request handling process and data generating process.

---

**Note:** Users need not to build RabbitMQ instance on the same server as Qlib-Server.

---

Build RabbitMQ according to the following processes:

- Import RabbitMQ signing key on your system:

```
echo 'deb http://www.rabbitmq.com/debian/ testing main' | sudo tee /etc/
↳apt/sources.list.d/rabbitmq.list
wget -O- https://www.rabbitmq.com/rabbitmq-release-signing-key.asc | sudo
↳apt-key add -
```

- Update apt cache and install RabbitMQ server on your system:

```
sudo apt-get update
sudo apt-get install rabbitmq-server
```

- Enable the RabbitMQ service and start it.

```
# Using Init -
sudo update-rc.d rabbitmq-server defaults
sudo service rabbitmq-server start
sudo service rabbitmq-server stop

# Using Systemctl -
```

(continues on next page)

(continued from previous page)

```
sudo systemctl enable rabbitmq-server
sudo systemctl start rabbitmq-server
sudo systemctl stop rabbitmq-server
```

- **Create admin user in RabbitMQ** By default RabbitMQ creates a username *guest* with password *guest*. Users can also create admin user in RabbitMQ:

```
sudo rabbitmqctl add_user admin <your password>
sudo rabbitmqctl set_user_tags admin administrator
sudo rabbitmqctl set_permissions -p / admin ".*" ".*" ".*"
```

- **Enable web management console** RabbitMQ also provides a web management console for managing the entire RabbitMQ. To enable web management console run following command. The web management console helps users with managing RabbitMQ server.

```
sudo rabbitmq-plugins enable rabbitmq_management
```

Visit *<your rabbitmq host>:15672* to manage your queue. Keep in mind your rabbitmq host and credentials. It will be used in qlib-server config.

## Build Redis

Qlib-Server needs redis to store and read some meta info as well as thread lock.

---

**Note:** Users need not to build redis instance on the same server as Qlib-Server.

---

Build redis according to the following processes:

- **Download the latest version of redis and install**

```
mkdir ~/redis
cd ~/redis
wget http://download.redis.io/releases/redis-5.0.4.tar.gz
tar -zxvf redis-5.0.4.tar.gz
cd redis-5.0.4
sudo make && make install
```

- **Start redis service**

```
/usr/local/bin/redis-server
```

The default port of redis is **6379**. Keep in mind your redis host and port. It will be used in qlib-server config.

## Build NFS

Before starting Qlib-Server, it's necessary to make sure the cache file directories are mounted (or at least ready to be mounted) to clients by configuring nfs service.

Build NFS according to the following processes:

- Install NFS service:

```
sudo apt-get install nfs-kernel-server
```

- Check if the nfs port is open:

```
netstat -tl
```

**Note:** By seeing `tcp 0 0 *:nfs *: LISTEN`, the nfs port is ready for listening. Restart the service to ensure it can be used:

```
sudo /etc/init.d/nfs-kernel-server restart
```

- Modify `/etc/exports` to give the directories ability to be mounted. To find out how the keywords like `rw` work and change them, please refer to nfs documents.

```
sudo echo '<your data directory> *(rw, sync, no_subtree_check, no_root_squash)' >> /etc/
↪ exports
```

Use `showmount` to view the exported directories.

### Build Qlib-Server

Users can choose one of the following two methods to build Qlib-Server:

- Build with Source Code
- Build with Dockerfile

### Build with Source Code

Build Qlib-Server with source code according to the following processes:

- Enter the Qlib-Server directory and run `python setup.py install`.
- Modify the `config.yaml` according to users' needs and configs.
- **Start using Qlib-server by running:**

```
cp config_template.yaml config.yaml
edit config.yaml # Please edit the server config.
python main.py -c config.yaml
```

**Warning:** Rabbitmq and redis configurations cannot be shared among multiple qlib-server instances

Eg:

```
In config_1.yaml, redis_db:1 task_queue: 'task_queue_1'
In config_2.yaml, redis_db:2 task_queue: 'task_queue_2'
-----
In config_1.yaml, redis_db:1 task_queue: 'task_queue_1' ×
In config_2.yaml, redis_db:1 task_queue: 'task_queue_1' ×
```

**Note:** The content of `config.yaml` is as follows

- ***provider\_uri*** Qlib data directory
- ***flask\_server*** Flask server host/ip, can be 0.0.0.0 or private ip
- ***flask\_port*** Data service port, with which the client port must be consistent to access server
- ***queue\_host*** RabbitMQ server ip/host
- ***queue\_user*** RabbitMQ user name
- ***queue\_pwd*** RabbitMQ password
- ***task\_queue*** Task queue of Qlib-Server, if rabbitmq serves multiple Qlib-Server s, this value cannot be repeated
- ***message\_queue*** Message queue of Qlib-Server, if rabbitmq serves multiple Qlib-Server s, this value cannot be repeated
- ***redis\_host*** Redis server host/ip
- ***redis\_port*** Redis server port
- ***redis\_task\_db*** Redis database name
- ***auto\_update*** Currently, this parameter is not used
- ***update\_time*** Currently, this parameter is not used
- ***client\_version*** The version of Qlib must be newer than *client\_version* to access the Qlib-Server
- ***server\_version*** The version of Qlib must be newer than *server\_version* to install or run Qlib-Server
- ***dataset\_cache\_dir\_name*** The name of the dataset cache directory, it is not recommended to modify
- ***features\_cache\_dir\_name*** The name of the features cache directory, it is not recommended to modify
- ***logging\_level*** Level control of Qlib-Server log
- ***logging\_config*** Log configuration, it is not recommended to modify

## Build from Dockerfile

Build Qlib-Server with Dockerfile according to the following processes:

- Install docker, please refer to [Docker Installation](#).
- Start using Qlib-Server by running:

```
git clone https://github.com/microsoft/qlib-server
cd qlib-server
sudo docker build -f docker_support/Dockerfile -t qlib-server \
  --build-arg QLIB_DATA=/data/stock_data/qlib_data \
    QUEUE_HOST=rabbitmq_server \
    QUEUE_USER=rabbitmq_user \
    QUEUE_PASS=rabbitmq_password \
    MESSAGE_QUEUE=message_queue \
    TASK_QUEUE=task_queue \
    REDIS_HOST=redis_server \
    REDIS_PORT=6379 \
    REDIS_DB=1 \
    FLASK_SERVER_HOST=127.0.0.1 \
    QLIB_CODE=/code \
sudo docker run -p 9710:9710 qlib-server
```

## 1.3 Using Qlib in Online Mode

### 1.3.1 Introduction

In the [Qlib Document](#), the Offline mode has been introduced. In addition to offline mode, users can use Qlib in Online mode.

The Online mode is designed to solve the following problems:

- Manage the data in a centralized way. Users don't have to manage data of different versions.
- Reduce the amount of cache to be generated.
- Make the data can be accessed in a remote way.

In Online mode, the data provided for Qlib will be managed in a centralized manner by Qlib-Server.

### 1.3.2 Using Qlib in Online Mode

Use Qlib in online mode according to the following steps:

- Open NFS Features in Qlib Client
- Initialize Qlib in online Mode

#### Opening NFS Features in Qlib Client

- If running on Linux, users need to install `nfs-common` on the client, execute:
- If running on Windows, do as follows.
  - Open Programs and Features.
  - Click Turn Windows features on or off.
  - Scroll down and check the option Services for NFS, then click OK

Reference address: <https://graspingtech.com/mount-nfs-share-windows-10/>

#### Initializing Qlib in online Mode

If users want to use Qlib in online mode, they can choose either of the following two methods to initialize Qlib:

- Initialize Qlib with configuration file
- Initialize Qlib with arguments

#### Configuration File

The content of configuration file is as follows.

```
calendar_provider:
  class: LocalCalendarProvider
  kwargs:
    remote: True
feature_provider:
  class: LocalFeatureProvider
```

(continues on next page)

(continued from previous page)

```

    kwargs:
        remote: True
expression_provider: LocalExpressionProvider
instrument_provider: ClientInstrumentProvider
dataset_provider: ClientDatasetProvider
provider: ClientProvider
expression_cache: null
dataset_cache: null
calendar_cache: null

provider_uri: 127.0.0.1:/
mount_path: /data/stock_data/qlib_data
auto_mount: True
flask_server: 127.0.0.1
flask_port: 9710

```

- **provider\_uri** nfs-server path; the format is host: data\_dir, for example: 127.0.0.1:/. If using Qlib in Local mode, it can be a local data directory.
- **mount\_path** local data directory, provider\_uri will be mounted to this directory
- **auto\_mount**: whether to automatically mount provider\_uri to mount\_path during qlib init; User can also mount it manually as follows.

---

**Note:** Automount requires sudo permission

---

- **flask\_server** data service host/ip
- **flask\_port** data service port

Initialize Qlib with parameters configuration file as follows.

```

import qlib
qlib.init_from_yaml_conf("qlib_clinet_config.yaml")
from qlib.data import D
ins = D.list_instruments(D.instruments("all"), as_list=True)

```

---

**Note:** If running Qlib on Windows, users should write correct **mount\_path**.

- In windows, mount path must be not exist path and root path,
  - correct format path eg: H, i...
  - error format path eg: C, C:/user/name, qlib\_data...

The configuration file can be:

```

...
...
provider_uri: 127.0.0.1:/
mount_path: H
auto_mount: True
flask_server: 127.0.0.1
flask_port: 9710

```

## Arguments

Initialize Qlib with arguments as follows.

```
import qlib

# qlib client config

ONLINE_CONFIG = {
    # data provider config
    "calendar_provider": {"class": "LocalCalendarProvider", "kwargs": {"remote": True}
    ↪},
    "instrument_provider": "ClientInstrumentProvider",
    "feature_provider": {"class": "LocalFeatureProvider", "kwargs": {"remote": True}},
    "expression_provider": "LocalExpressionProvider",
    "dataset_provider": "ClientDatasetProvider",
    "provider": "ClientProvider",
    # config it in user's own code
    "provider_uri": "127.0.0.1:/",
    # cache
    # Using parameter 'remote' to announce the client is using server_cache, and the
    ↪writing access will be disabled.
    "expression_cache": None,
    "dataset_cache": None,
    "calendar_cache": None,
    "mount_path": "/data/stock_data/qlib_data",
    "auto_mount": True, # The nfs is already mounted on our server[auto_mount:
    ↪False].
    "flask_server": "127.0.0.1",
    "flask_port": 9710,
    "region": "cn",
}

qlib.init(**client_config)
ins = D.list_instruments(D.instruments("all"), as_list=True)
```

---

**Note:** If running Qlib on Windows, users should write correct **mount\_path**.

The arguments can be:

```
ONLINE_CONFIG = {
    ...
    ...
    "mount_path": "H",
    "auto_mount": True,
    "flask_server": "127.0.0.1",
    "flask_port": 9710,
    "region": "cn",
}
```

---

## 1.4 Changelog

Here you can see the full list of changes between each QLibServer release.



### 1.4.1 Version 0.1.0

This is the initial release of QLibServer.

### 1.4.2 Version 0.1.1

- Fix multi-process log bug, using `logging.handlers.QueueHandler`
- Modify to create a `rabbitmq connection` and `channel` for each process, before sharing a one that causes `basic_ack` to be unsuccessful
- Add `max_process` to `config`
- The log is set through the configuration file, refer to `config.yaml.example`

### 1.4.3 Version 0.1.2

- Format PEP8