Platform API

FutureHouse Platform API Documentation

pypi package 0.3.17 ↗ License Apache 2.0 python 3.11 | 3.12

Documentation and tutorials for futurehouse-client, a client for interacting with endpoints of the FutureHouse platform.

Installation

```
uv pip install futurehouse-client
```

Quickstart

```python
from futurehouse_client import FutureHouseClient, JobNames
from pathlib import Path
from aviary.core import DummyEnv
import ldp

client = FutureHouseClient(
    api_key="your_api_key",
)

task_data = {
    "name": JobNames.CROW,
    "query": "Which neglected diseases had a treatment developed by
artificial intelligence?",
}

task_response = client.run_tasks_until_done(task_data)
```

A quickstart example can be found in the client_notebook.ipynb ↗ file, where we show how to submit and retrieve a task, pass runtime configuration to the agent, and ask follow-up questions to the previous task.

Functionalities

FutureHouse client implements a RestClient (called `FutureHouseClient`) with the following functionalities:

- Simple task running: `run_tasks_until_done(TaskRequest)` or `await arun_tasks_until_done(TaskRequest)`
- Asynchronous tasks: `get_task(task_id)` or `aget_task(task_id)` and `create_task(TaskRequest)` or `acreate_task(TaskRequest)`

To create a `FutureHouseClient`, you need to pass a FutureHouse platform api key (see Authentication):

```python
from futurehouse_client import FutureHouseClient

client = FutureHouseClient(
    api_key="your_api_key",
)
```

Authentication

In order to use the `FutureHouseClient`, you need to authenticate yourself. Authentication is done by providing an API key, which can be obtained directly from your profile page in the FutureHouse platform ↗.

Simple task running

In the FutureHouse platform, we define the deployed combination of an agent and an environment as a `job`. To invoke a job, we need to submit a `task` (also called a `query`) to it. `FutureHouseClient` can be used to submit tasks/queries to available jobs in the FutureHouse platform. Using a `FutureHouseClient` instance, you can submit tasks to the platform by calling the `create_task` method, which receives a `TaskRequest` (or a dictionary with `kwargs`) and returns the task id.
Aiming to make the submission of tasks as simple as possible, we have created a `JobNames` `enum` that contains the available task types.

The available supported jobs are:

Alias	Job Name	Task type	Description
`JobNames.CROW`	`job-futurehouse-paperqa2`	Fast Search	Ask a question of scientific data sources, and receive a high-accuracy, cited response. Built with PaperQA2 ↗.
`JobNames.FALCON`	`job-futurehouse-paperqa2-deep`	Deep Search	Use a plethora of sources to deeply research. Receive a detailed, structured report as a response.
`JobNames.OWL`	`job-futurehouse-hasanyone`	Precedent Search	Formerly known as HasAnyone, query if

			anyone has ever done something in science.
`JobNames.PHOENIX`	`job-futurehouse-phoenix`	Chemistry Tasks	A new iteration of ChemCrow, Phoenix uses cheminformatics tools to do chemistry. Good for planning synthesis and design of new molecules.
`JobNames.DUMMY`	`job-futurehouse-dummy`	Dummy Task	This is a dummy task. Mainly for testing purposes.

Using `JobNames`, the task submission looks like this:

```
from futurehouse_client import FutureHouseClient, JobNames

client = FutureHouseClient(
    api_key="your_api_key",
)

task_data = {
    "name": JobNames.OWL,
    "query": "Has anyone tested therapeutic exerkines in humans or
NHPs?",
}

task_response = client.run_tasks_until_done(task_data)

print(task_response.answer)
```

Or if running async code:

```
import asyncio
from futurehouse_client import FutureHouseClient, JobNames


async def main():
    client = FutureHouseClient(
        api_key="your_api_key",
    )

    task_data = {
        "name": JobNames.OWL,
        "query": "Has anyone tested therapeutic exerkines in humans or
NHPs?",
    }

    task_response = await client.arun_tasks_until_done(task_data)
    print(task_response.answer)
    return task_id


# For Python 3.7+
if __name__ == "__main__":
    task_id = asyncio.run(main())
```

Note that in either the sync or the async code, collections of tasks can be given to the client to run them in a batch:

```
import asyncio
from futurehouse_client import FutureHouseClient, JobNames


async def main():
    client = FutureHouseClient(
        api_key="your_api_key",
    )

    task_data = [{
        "name": JobNames.OWL,
        "query": "Has anyone tested therapeutic exerkines in humans or
NHPs?",
    },
    {
        "name": JobNames.CROW,
        "query": "Are there any clinically validated therapeutic
exerkines for humans?",
    }
    ]

    task_responses = await client.arun_tasks_until_done(task_data)
    print(task_responses[0].answer)
    print(task_responses[1].answer)
    return task_id


# For Python 3.7+
if __name__ == "__main__":
    task_id = asyncio.run(main())
```

`TaskRequest` can also be used to submit jobs and it has the following fields:

Field	Type	Description
id	UUID	Optional job identifier. A UUID will be generated if not provided
name	str	Name of the job to execute eg. `job-futurehouse-paperqa2`, or using the `JobNames` for convenience: `JobNames.CROW`

query	str	Query or task to be executed by the job
runtime_config	RuntimeConfig	Optional runtime parameters

`runtime_config` can receive a `AgentConfig` object with the desired kwargs. Check the available `AgentConfig` fields in the LDP documentation ↗. Besides the `AgentConfig` object, we can also pass `timeout` and `max_steps` to limit the execution time and the number of steps the agent can take.

```
from futurehouse_client import FutureHouseClient, JobNames
from futurehouse_client.models.app import TaskRequest

client = FutureHouseClient(
    api_key="your_api_key",
)

task_response = client.run_tasks_until_done(
    TaskRequest(
        name=JobNames.OWL,
        query="Has anyone tested therapeutic exerkines in humans or
NHPs?",
    )
)

print(task_response.answer)
```

A `TaskResponse` will be returned from using our agents. For Owl, Crow, and Falcon, we default to a subclass, `PQATaskResponse` which has some key attributes:

Field	Type	Description
answer	str	Answer to your query.
formatted_answer	str	Specially formatted answer with references.
has_successful_answer	bool	Flag for whether the agent was able to find a good answer to your query or not.

If using the `verbose` setting, much more data can be pulled down from your `TaskResponse`, which will exist across all agents (not just Owl, Crow, and Falcon).

```
from futurehouse_client import FutureHouseClient, JobNames
from futurehouse_client.models.app import TaskRequest

client = FutureHouseClient(
    api_key="your_api_key",
)

task_response = client.run_tasks_until_done(
    TaskRequest(
        name=JobNames.OWL,
        query="Has anyone tested therapeutic exerkines in humans or
NHPs?",
    ),
    verbose=True,
)

print(task_response.environment_frame)
```

In that case, a `TaskResponseVerbose` will have the following fields:

Field	Type	Description
agent_state	dict	Large object with all agent states during the progress of your task.
environment_frame	dict	Large nested object with all environment data, for PQA environments it includes contexts, paper metadata, and answers.
metadata	dict	Extra metadata about your query.

Task Continuation

Once a task is submitted and the answer is returned, FutureHouse platform allow you to ask follow-up questions to the previous task.
It is also possible through the platform API.
To accomplish that, we can use the `runtime_config` we discussed in the Simple task running section.

```
from futurehouse_client import FutureHouseClient, JobNames

client = FutureHouseClient(
    api_key="your_api_key",
)

task_data = {"name": JobNames.CROW, "query": "How many species of birds
are there?"}

task_id = client.create_task(task_data)

continued_task_data = {
    "name": JobNames.CROW,
    "query": "From the previous answer, specifically,how many species
of crows are there?",
    "runtime_config": {"continued_task_id": task_id},
}

task_result = client.run_tasks_until_done(continued_task_data)
```

Asynchronous tasks

Sometimes you may want to submit many jobs, while querying results at a later time. In this way you can do other things while waiting for a response. The platform API supports this as well rather than waiting for a result.

```
from futurehouse_client import FutureHouseClient

client = FutureHouseClient(
    api_key="your_api_key",
)

task_data = {"name": JobNames.CROW, "query": "How many species of birds
are there?"}

task_id = client.create_task(task_data)

# move on to do other things

task_status = client.get_task(task_id)
```

`task_status` contains information about the task. For instance, its `status`, `task`, `environment_name` and `agent_name`, and other fields specific to the job. You can continually query the status until it's `success` before moving on.

docs

client_notebook

FutureHouse platform client usage example

```
from futurehouse_client import FutureHouseClient, JobNames
from futurehouse_client.models import (
    RuntimeConfig,
    TaskRequest,
)
from ldp.agent import AgentConfig
```

Client instantiation

Here we use `auth_type=AuthType.API_KEY` to authenticate with the platform.
Please log in to the platform and go to your user settings to get your API key.

```
client = FutureHouseClient(
    api_key="your-api-key",
)
```

Submit a task to an available futurehouse job

In the futurehouse platform, we refer to the deployed combination of agent and environment as a `job`.
Submitting task to a futurehouse job is done by calling the `create_task` method, which receives a `TaskRequest` object.

```
task_data = TaskRequest(
    name=JobNames.from_string("crow"),
    query="What is the molecule known to have the greatest solubility
in water?",
)
task_response = client.run_tasks_until_done(task_data)

print(f"Job status: {task_response.status}")
print(f"Job answer: \n{task_response.formatted_answer}")
```

You can also pass a `runtime_config` to the `create_task` method, which will be used to configure the agent on runtime.
Here, we will define a agent configuration and include it in the `TaskRequest`. This agent is used to decide the next action to take.
We will also use the `max_steps` parameter to limit the number of steps the agent will take.

```
agent = AgentConfig(
    agent_type="SimpleAgent",
    agent_kwargs={
        "model": "gpt-4o",
        "temperature": 0.0,
    },
)
task_data = TaskRequest(
    name=JobNames.CROW,
    query="How many moons does earth have?",
    runtime_config=RuntimeConfig(agent=agent, max_steps=10),
)
task_response = client.run_tasks_until_done(task_data)

print(f"Job status: {task_response.status}")
print(f"Job answer: \n{task_response.formatted_answer}")
```

Continue a job

The platform allows to ask follow-up questions to the previous job.
To accomplish that, we can use the `runtime_config` to pass the `task_id` of the previous task.

Notice that `create_task` accepts both a `TaskRequest` object and a dictionary with keywords arguments.

```
task_data = TaskRequest(
    name=JobNames.CROW, query="How many species of birds are there?"
)

task_response = client.run_tasks_until_done(task_data)

print(f"First job status: {task_response.status}")
print(f"First job answer: \n{task_response.formatted_answer}")
```

```
continued_job_data = {
    "name": JobNames.CROW,
    "query": (
        "From the previous answer, specifically,how many species of
crows are there?"
    ),
    "runtime_config": {"continued_job_id": task_response.task_id},
}

continued_task_response =
client.run_tasks_until_done(continued_job_data)


print(f"Continued job status: {continued_task_response.status}")
print(f"Continued job answer:
\n{continued_task_response.formatted_answer}")
```