## Lab: Prefect - Tasks

A task is any Python function decorated with a `@task` decorator called within a flow. You can think of a flow as a recipe for connecting a known sequence of tasks together. Tasks, and the dependencies between them, are displayed in the flow run graph, enabling you to break down a complex flow into something you can observe, understand and control at a more granular level. When a function becomes a task, it can be executed concurrently and its return value can be cached.

Flows and tasks share some common features:

- Both are defined easily using their respective decorator, which accepts settings for that flow / task.
- Each can be given a name, description and tags for organization and bookkeeping.
- Both provide functionality for retries, timeouts, and other hooks to handle failure and completion events.

Network calls (such as our `GET` requests to the GitHub API) are particularly useful as tasks because they take advantage of task features such as retries, caching, and concurrency.

**Tasks must be called from flows**

All tasks must be called from within a flow. Tasks may not call other tasks directly.

**When to use tasks**

Not all functions in a flow need be tasks. Use them only when their features are useful.

Let's take our flow from before and move the request into a task:

Update following code in python file `repo_info.py`:

```
import httpx
from prefect import flow, task


@task
def get_url(url: str, params: dict = None):
    response = httpx.get(url, params=params)
    response.raise_for_status()
    return response.json()


@flow(retries=3, retry_delay_seconds=5, log_prints=True)
def get_repo_info(repo_name: str = "PrefectHQ/prefect"):
    url = f"https://api.github.com/repos/{repo_name}"
    repo_stats = get_url(url)
    print(f"{repo_name} repository statistics 🤓:")
    print(f"Stars 🌠 : {repo_stats['stargazers_count']}")
    print(f"Forks 🍴 : {repo_stats['forks_count']}")

if __name__ == "__main__":
    get_repo_info()
```

Running the flow in your terminal will result in something like this:

```
09:55:55.412 | INFO    | prefect.engine - Created flow run 'great-ammonite' for flow
'get-repo-info'
09:55:55.499 | INFO    | Flow run 'great-ammonite' - Created task run 'get_url-0' for
```

```
task 'get_url'
09:55:55.500 | INFO    | Flow run 'great-ammonite' - Executing 'get_url-0'
immediately...
09:55:55.825 | INFO    | Task run 'get_url-0' - Finished in state Completed()
09:55:55.827 | INFO    | Flow run 'great-ammonite' - PrefectHQ/prefect repository
statistics 🤓:
09:55:55.827 | INFO    | Flow run 'great-ammonite' - Stars 🌠 : 12157
09:55:55.827 | INFO    | Flow run 'great-ammonite' - Forks 🍴 : 1251
09:55:55.849 | INFO    | Flow run 'great-ammonite' - Finished in state Completed('All
states completed.')
```

And you should now see this task run tracked in the UI as well.

### Caching

Tasks support the ability to cache their return value. Caching allows you to efficiently reuse results of tasks that may be expensive to reproduce with every flow run, or reuse cached results if the inputs to a task have not changed.

To enable caching, specify a cache_key_fn — a function that returns a cache key — on your task. You may optionally provide a cache_expiration timedelta indicating when the cache expires. You can define a task that is cached based on its inputs by using the Prefect task_input_hash. Let's add caching to our get_url task:

```
import httpx
from datetime import timedelta
from prefect import flow, task, get_run_logger
from prefect.tasks import task_input_hash


@task(cache_key_fn=task_input_hash,
      cache_expiration=timedelta(hours=1),
      )
def get_url(url: str, params: dict = None):
    response = httpx.get(url, params=params)
    response.raise_for_status()
    return response.json()
```

You can test this caching behavior by using a personal repository as your workflow parameter - give it a star, or remove a star and see how the output of this task changes (or doesn't) by running your flow multiple times.

#### Task results and caching

Task results are cached in memory during a flow run and persisted to your home directory by default. Prefect Cloud only stores the cache key, not the data itself.

### Concurrency

Tasks enable concurrency, allowing you to execute multiple tasks asynchronously. This concurrency can greatly enhance the efficiency and performance of your workflows. Let's expand our script to calculate the average open issues per user. This will require making more requests:

Update following code in python file `repo_info.py`:

```
import httpx
from datetime import timedelta
from prefect import flow, task
from prefect.tasks import task_input_hash


@task(cache_key_fn=task_input_hash, cache_expiration=timedelta(hours=1))
def get_url(url: str, params: dict = None):
    response = httpx.get(url, params=params)
    response.raise_for_status()
    return response.json()


def get_open_issues(repo_name: str, open_issues_count: int, per_page: int = 100):
    issues = []
    pages = range(1, -(open_issues_count // -per_page) + 1)
    for page in pages:
        issues.append(
            get_url(
                f"https://api.github.com/repos/{repo_name}/issues",
                params={"page": page, "per_page": per_page, "state": "open"},
            )
        )
    return [i for p in issues for i in p]


@flow(retries=3, retry_delay_seconds=5, log_prints=True)
def get_repo_info(repo_name: str = "PrefectHQ/prefect"):
    repo_stats = get_url(f"https://api.github.com/repos/{repo_name}")
    issues = get_open_issues(repo_name, repo_stats["open_issues_count"])
    issues_per_user = len(issues) / len(set([i["user"]["id"] for i in issues]))
    print(f"{repo_name} repository statistics 🤓:")
    print(f"Stars 🌠 : {repo_stats['stargazers_count']}")
    print(f"Forks 🍴 : {repo_stats['forks_count']}")
    print(f"Average open issues per user 💌 : {issues_per_user:.2f}")


if __name__ == "__main__":
    get_repo_info()
```

Now we're fetching the data we need, but the requests are happening sequentially. Tasks expose a submit method that changes the execution from sequential to concurrent. In our specific example, we also need to use the result method because we are unpacking a list of return values:

```
def get_open_issues(repo_name: str, open_issues_count: int, per_page: int = 100):
    issues = []
    pages = range(1, -(open_issues_count // -per_page) + 1)
    for page in pages:
        issues.append(
            get_url.submit(
                f"https://api.github.com/repos/{repo_name}/issues",
                params={"page": page, "per_page": per_page, "state": "open"},
            )
        )
    return [i for p in issues for i in p.result()]
```

The logs show that each task is running concurrently:

```
12:45:28.241 | INFO    | prefect.engine - Created flow run 'intrepid-coua' for flow 'get-repo-info'
12:45:28.311 | INFO    | Flow run 'intrepid-coua' - Created task run 'get_url-0' for task 'get_url'
12:45:28.312 | INFO    | Flow run 'intrepid-coua' - Executing 'get_url-0' immediately...
12:45:28.543 | INFO    | Task run 'get_url-0' - Finished in state Completed()
12:45:28.583 | INFO    | Flow run 'intrepid-coua' - Created task run 'get_url-1' for task 'get_url'
12:45:28.584 | INFO    | Flow run 'intrepid-coua' - Submitted task run 'get_url-1' for execution.
12:45:28.594 | INFO    | Flow run 'intrepid-coua' - Created task run 'get_url-2' for task 'get_url'
12:45:28.594 | INFO    | Flow run 'intrepid-coua' - Submitted task run 'get_url-2' for execution.
12:45:28.609 | INFO    | Flow run 'intrepid-coua' - Created task run 'get_url-4' for task 'get_url'
12:45:28.610 | INFO    | Flow run 'intrepid-coua' - Submitted task run 'get_url-4' for execution.
12:45:28.624 | INFO    | Flow run 'intrepid-coua' - Created task run 'get_url-5' for task 'get_url'
12:45:28.625 | INFO    | Flow run 'intrepid-coua' - Submitted task run 'get_url-5' for execution.
12:45:28.640 | INFO    | Flow run 'intrepid-coua' - Created task run 'get_url-6' for task 'get_url'
12:45:28.641 | INFO    | Flow run 'intrepid-coua' - Submitted task run 'get_url-6' for execution.
12:45:28.708 | INFO    | Flow run 'intrepid-coua' - Created task run 'get_url-3' for task 'get_url'
12:45:28.708 | INFO    | Flow run 'intrepid-coua' - Submitted task run 'get_url-3' for execution.
12:45:29.096 | INFO    | Task run 'get_url-6' - Finished in state Completed()
12:45:29.565 | INFO    | Task run 'get_url-2' - Finished in state Completed()
12:45:29.721 | INFO    | Task run 'get_url-5' - Finished in state Completed()
12:45:29.749 | INFO    | Task run 'get_url-4' - Finished in state Completed()
12:45:29.801 | INFO    | Task run 'get_url-3' - Finished in state Completed()
12:45:29.817 | INFO    | Task run 'get_url-1' - Finished in state Completed()
12:45:29.820 | INFO    | Flow run 'intrepid-coua' - PrefectHQ/prefect repository statistics 🤓:
12:45:29.820 | INFO    | Flow run 'intrepid-coua' - Stars 🌠 : 12159
12:45:29.821 | INFO    | Flow run 'intrepid-coua' - Forks 🍴 : 1251
Average open issues per user 💌 : 2.27
12:45:29.838 | INFO    | Flow run 'intrepid-coua' - Finished in state Completed('All states completed.')
```

## Subflows

Not only can you call tasks within a flow, but you can also call other flows! Child flows are called subflows and allow you to efficiently manage, track, and version common multi-task logic.

Subflows are a great way to organize your workflows and offer more visibility within the UI.

Let's add a `flow` decorator to our get_open_issues function:

```
@flow
def get_open_issues(repo_name: str, open_issues_count: int, per_page: int = 100):
    issues = []
    pages = range(1, -(open_issues_count // -per_page) + 1)
    for page in pages:
        issues.append(
            get_url.submit(
                f"https://api.github.com/repos/{repo_name}/issues",
                params={"page": page, "per_page": per_page, "state": "open"},
            )
        )
    return [i for p in issues for i in p.result()]
```

Whenever we run the parent flow, a new run will be generated for related functions within that as well. Not only is this run tracked as a subflow run of the main flow, but you can also inspect it independently in the UI!