Background Tasks

You can define background tasks to be run *after* returning a response.

This is useful for operations that need to happen after a request, but that the client doesn't really have to be waiting for the operation to complete before receiving the response.

This includes, for example:

- Email notifications sent after performing an action:
  - As connecting to an email server and sending an email tends to be "slow" (several seconds), you can return the response right away and send the email notification in the background.
- Processing data:
  - For example, let's say you receive a file that must go through a slow process, you can return a response of "Accepted" (HTTP 202) and process it in the background.

Using `BackgroundTasks`

First, import `BackgroundTasks` and define a parameter in your *path operation function* with a type declaration of `BackgroundTasks` :

```
{!../../../docs_src/background_tasks/tutorial001.py!}
```

FastAPI will create the object of type `BackgroundTasks` for you and pass it as that parameter.

Create a task function

Create a function to be run as the background task.

It is just a standard function that can receive parameters.

It can be an `async def` or normal `def` function, **FastAPI** will know how to handle it correctly.

In this case, the task function will write to a file (simulating sending an email).

And as the write operation doesn't use `async` and `await` , we define the function with normal `def` :

```
{!../../../docs_src/background_tasks/tutorial001.py!}
```

Add the background task

Inside of your *path operation function*, pass your task function to the *background tasks* object with the method `.add_task()` :

```
{!../../../docs_src/background_tasks/tutorial001.py!}
```

`.add_task()` receives as arguments:

- A task function to be run in the background (`write_notification`).
- Any sequence of arguments that should be passed to the task function in order (`email`).

- Any keyword arguments that should be passed to the task function (`message="some notification"`).

Dependency Injection

Using `BackgroundTasks` also works with the dependency injection system, you can declare a parameter of type `BackgroundTasks` at multiple levels: in a *path operation function*, in a dependency (dependable), in a sub-dependency, etc.

FastAPI knows what to do in each case and how to re-use the same object, so that all the background tasks are merged together and are run in the background afterwards:

=== "Python 3.6 and above"

~~~
```Python hl_lines="13  15  22  25"
{!> ../../../docs_src/background_tasks/tutorial002.py!}
```
~~~

=== "Python 3.10 and above"

~~~
```Python hl_lines="11  13  20  23"
{!> ../../../docs_src/background_tasks/tutorial002_py310.py!}
```
~~~

In this example, the messages will be written to the `log.txt` file *after* the response is sent.

If there was a query in the request, it will be written to the log in a background task.

And then another background task generated at the *path operation function* will write a message using the `email` path parameter.

Technical Details

The class `BackgroundTasks` comes directly from `starlette.background`.

It is imported/included directly into FastAPI so that you can import it from `fastapi` and avoid accidentally importing the alternative `BackgroundTask` (without the `s` at the end) from `starlette.background`.

By only using `BackgroundTasks` (and not `BackgroundTask`), it's then possible to use it as a *path operation function* parameter and have **FastAPI** handle the rest for you, just like when using the `Request` object directly.

It's still possible to use `BackgroundTask` alone in FastAPI, but you have to create the object in your code and return a Starlette `Response` including it.

You can see more details in Starlette's official docs for Background Tasks.

Caveat

If you need to perform heavy background computation and you don't necessarily need it to be run by the same process (for example, you don't need to share memory, variables, etc), you might benefit from using other bigger tools like Celery.

They tend to require more complex configurations, a message/job queue manager, like RabbitMQ or Redis, but they allow you to run background tasks in multiple processes, and especially, in multiple servers.

To see an example, check the Project Generators{.internal-link target=_blank}, they all include Celery already configured.

But if you need to access variables and objects from the same **FastAPI** app, or you need to perform small background tasks (like sending an email notification), you can simply just use `BackgroundTasks`.

Recap

Import and use `BackgroundTasks` with parameters in *path operation functions* and dependencies to add background tasks.