Bigger Applications - Multiple Files

If you are building an application or a web API, it's rarely the case that you can put everything on a single file.

FastAPI provides a convenience tool to structure your application while keeping all the flexibility.

!!! info If you come from Flask, this would be the equivalent of Flask's Blueprints.

An example file structure

Let's say you have a file structure like this:

```
.
├── app
│   ├── __init__.py
│   ├── main.py
│   ├── dependencies.py
│   └── routers
│   │   ├── __init__.py
│   │   ├── items.py
│   │   └── users.py
│   └── internal
│       ├── __init__.py
│       └── admin.py
```

!!! tip There are several `__init__.py` files: one in each directory or subdirectory.

~~~
This is what allows importing code from one file into another.

For example, in `app/main.py` you could have a line like:

```
from app.routers import items
```
~~~

- The `app` directory contains everything. And it has an empty file `app/__init__.py`, so it is a "Python package" (a collection of "Python modules"): `app`.
- It contains an `app/main.py` file. As it is inside a Python package (a directory with a file `__init__.py`), it is a "module" of that package: `app.main`.
- There's also an `app/dependencies.py` file, just like `app/main.py`, it is a "module": `app.dependencies`.
- There's a subdirectory `app/routers/` with another file `__init__.py`, so it's a "Python subpackage": `app.routers`.
- The file `app/routers/items.py` is inside a package, `app/routers/`, so, it's a submodule: `app.routers.items`.
- The same with `app/routers/users.py`, it's another submodule: `app.routers.users`.
- There's also a subdirectory `app/internal/` with another file `__init__.py`, so it's another "Python subpackage": `app.internal`.
- And the file `app/internal/admin.py` is another submodule: `app.internal.admin`.

The same file structure with comments:

```
.
├── app                  # "app" is a Python package
│   ├── __init__.py      # this file makes "app" a "Python package"
│   ├── main.py          # "main" module, e.g. import app.main
│   ├── dependencies.py  # "dependencies" module, e.g. import app.dependencies
│   └── routers          # "routers" is a "Python subpackage"
│   │   ├── __init__.py  # makes "routers" a "Python subpackage"
│   │   ├── items.py     # "items" submodule, e.g. import app.routers.items
│   │   └── users.py     # "users" submodule, e.g. import app.routers.users
│   └── internal         # "internal" is a "Python subpackage"
│       ├── __init__.py  # makes "internal" a "Python subpackage"
│       └── admin.py     # "admin" submodule, e.g. import app.internal.admin
```

`APIRouter`

Let's say the file dedicated to handling just users is the submodule at `/app/routers/users.py`.

You want to have the *path operations* related to your users separated from the rest of the code, to keep it organized.

But it's still part of the same **FastAPI** application/web API (it's part of the same "Python Package").

You can create the *path operations* for that module using `APIRouter`.

Import `APIRouter`

You import it and create an "instance" the same way you would with the class `FastAPI`:

```
{!../../../docs_src/bigger_applications/app/routers/users.py!}
```

Path operations with `APIRouter`

And then you use it to declare your *path operations*.

Use it the same way you would use the `FastAPI` class:

```
{!../../../docs_src/bigger_applications/app/routers/users.py!}
```

You can think of `APIRouter` as a "mini `FastAPI`" class.

All the same options are supported.

All the same `parameters`, `responses`, `dependencies`, `tags`, etc.

!!! tip In this example, the variable is called `router`, but you can name it however you want.

We are going to include this `APIRouter` in the main `FastAPI` app, but first, let's check the dependencies and another `APIRouter`.

Dependencies

We see that we are going to need some dependencies used in several places of the application.

So we put them in their own `dependencies` module (`app/dependencies.py`).

We will now use a simple dependency to read a custom `X-Token` header:

```
{!../../../docs_src/bigger_applications/app/dependencies.py!}
```

!!! tip We are using an invented header to simplify this example.

```
But in real cases you will get better results using the integrated [Security
utilities](./security/index.md){.internal-link target=_blank}.
```

Another module with `APIRouter`

Let's say you also have the endpoints dedicated to handling "items" from your application in the module at `app/routers/items.py` .

You have *path operations* for:

- `/items/`
- `/items/{item_id}`

It's all the same structure as with `app/routers/users.py` .

But we want to be smarter and simplify the code a bit.

We know all the *path operations* in this module have the same:

- Path `prefix` : `/items` .
- `tags` : (just one tag: `items`).
- Extra `responses` .
- `dependencies` : they all need that `X-Token` dependency we created.

So, instead of adding all that to each *path operation*, we can add it to the `APIRouter` .

```
{!../../../docs_src/bigger_applications/app/routers/items.py!}
```

As the path of each *path operation* has to start with `/` , like in:

```
@router.get("/{item_id}")
async def read_item(item_id: str):
    ...
```

...the prefix must not include a final `/` .

So, the prefix in this case is `/items` .

We can also add a list of `tags` and extra `responses` that will be applied to all the *path operations* included in this router.

And we can add a list of `dependencies` that will be added to all the *path operations* in the router and will be executed/solved for each request made to them.

!!! tip Note that, much like [dependencies in *path operation decorators*](){.internal-link target=_blank}, no value will be passed to your *path operation function*.

The end result is that the item paths are now:

- `/items/`
- `/items/{item_id}`

...as we intended.

- They will be marked with a list of tags that contain a single string `"items"`.
    - These "tags" are especially useful for the automatic interactive documentation systems (using OpenAPI).
- All of them will include the predefined `responses`.
- All these *path operations* will have the list of `dependencies` evaluated/executed before them.
    - If you also declare dependencies in a specific *path operation*, **they will be executed too**.
    - The router dependencies are executed first, then the [`dependencies` in the decorator](){.internal-link target=_blank}, and then the normal parameter dependencies.
    - You can also add [`Security` dependencies with `scopes`](){.internal-link target=_blank}.

!!! tip Having `dependencies` in the `APIRouter` can be used, for example, to require authentication for a whole group of *path operations*. Even if the dependencies are not added individually to each one of them.

!!! check The `prefix`, `tags`, `responses`, and `dependencies` parameters are (as in many other cases) just a feature from **FastAPI** to help you avoid code duplication.

Import the dependencies

This codes lives in the module `app.routers.items`, the file `app/routers/items.py`.

And we need to get the dependency function from the module `app.dependencies`, the file `app/dependencies.py`.

So we use a relative import with `..` for the dependencies:

```
{!../../../docs_src/bigger_applications/app/routers/items.py!}
```

How relative imports work

!!! tip If you know perfectly how imports work, continue to the next section below.

A single dot `.`, like in:

```
from .dependencies import get_token_header
```

would mean:

- Starting in the same package that this module (the file `app/routers/items.py`) lives in (the directory `app/routers/`)...
- find the module `dependencies` (an imaginary file at `app/routers/dependencies.py`)...

* and from it, import the function `get_token_header`.

But that file doesn't exist, our dependencies are in a file at `app/dependencies.py`.

Remember how our app/file structure looks like:

The two dots `..`, like in:

```
from ..dependencies import get_token_header
```

mean:

* Starting in the same package that this module (the file `app/routers/items.py`) lives in (the directory `app/routers/`)...
* go to the parent package (the directory `app/`)...
* and in there, find the module `dependencies` (the file at `app/dependencies.py`)...
* and from it, import the function `get_token_header`.

That works correctly! 🎉

The same way, if we had used three dots `...`, like in:

```
from ...dependencies import get_token_header
```

that would mean:

* Starting in the same package that this module (the file `app/routers/items.py`) lives in (the directory `app/routers/`)...
* go to the parent package (the directory `app/`)...
* then go to the parent of that package (there's no parent package, `app` is the top level 😱)...
* and in there, find the module `dependencies` (the file at `app/dependencies.py`)...
* and from it, import the function `get_token_header`.

That would refer to some package above `app/`, with its own file `__init__.py`, etc. But we don't have that. So, that would throw an error in our example. 🚨

But now you know how it works, so you can use relative imports in your own apps no matter how complex they are. 🤓

Add some custom `tags`, `responses`, and `dependencies`

We are not adding the prefix `/items` nor the `tags=["items"]` to each *path operation* because we added them to the `APIRouter`.

But we can still add *more* `tags` that will be applied to a specific *path operation*, and also some extra `responses` specific to that *path operation*:

```
{!../../../docs_src/bigger_applications/app/routers/items.py!}
```

!!! tip This last path operation will have the combination of tags: `["items", "custom"]`.

```
And it will also have both responses in the documentation, one for `404` and one for
`403`.
```

The main `FastAPI`

Now, let's see the module at `app/main.py`.

Here's where you import and use the class `FastAPI`.

This will be the main file in your application that ties everything together.

And as most of your logic will now live in its own specific module, the main file will be quite simple.

Import `FastAPI`

You import and create a `FastAPI` class as normally.

And we can even declare [global dependencies]{.internal-link target=_blank} that will be combined with the dependencies for each `APIRouter`:

```
{!../../../docs_src/bigger_applications/app/main.py!}
```

Import the `APIRouter`

Now we import the other submodules that have `APIRouter`s:

```
{!../../../docs_src/bigger_applications/app/main.py!}
```

As the files `app/routers/users.py` and `app/routers/items.py` are submodules that are part of the same Python package `app`, we can use a single dot `.` to import them using "relative imports".

How the importing works

The section:

```
from .routers import items, users
```

Means:

- Starting in the same package that this module (the file `app/main.py`) lives in (the directory `app/`)...
- look for the subpackage `routers` (the directory at `app/routers/`)...
- and from it, import the submodule `items` (the file at `app/routers/items.py`) and `users` (the file at `app/routers/users.py`)...

The module `items` will have a variable `router` (`items.router`). This is the same one we created in the file `app/routers/items.py`, it's an `APIRouter` object.

And then we do the same for the module `users`.

We could also import them like:

```
from app.routers import items, users
```

!!! info The first version is a "relative import":

~~~
```Python
from .routers import items, users
~~~

The second version is an "absolute import":

```Python
from app.routers import items, users
```

To learn more about Python Packages and Modules, read <a
href="https://docs.python.org/3/tutorial/modules.html" class="external-link"
target="_blank">the official Python documentation about Modules</a>.
```

Avoid name collisions

We are importing the submodule `items` directly, instead of importing just its variable `router`.

This is because we also have another variable named `router` in the submodule `users`.

If we had imported one after the other, like:

```
from .routers.items import router
from .routers.users import router
```

The `router` from `users` would overwrite the one from `items` and we wouldn't be able to use them at the same time.

So, to be able to use both of them in the same file, we import the submodules directly:

```
{!../../../docs_src/bigger_applications/app/main.py!}
```

Include the `APIRouter`s for `users` and `items`

Now, let's include the `router`s from the submodules `users` and `items`:

```
{!../../../docs_src/bigger_applications/app/main.py!}
```

!!! info `users.router` contains the `APIRouter` inside of the file `app/routers/users.py`.

```
And `items.router` contains the `APIRouter` inside of the file `app/routers/items.py`.
```

With `app.include_router()` we can add each `APIRouter` to the main `FastAPI` application.

It will include all the routes from that router as part of it.
```

!!! note "Technical Details" It will actually internally create a *path operation* for each *path operation* that was declared in the `APIRouter`.

```
So, behind the scenes, it will actually work as if everything was the same single app.
```

!!! check You don't have to worry about performance when including routers.

```
This will take microseconds and will only happen at startup.

So it won't affect performance. ⚡
```

Include an `APIRouter` with a custom `prefix`, `tags`, `responses`, and `dependencies`

Now, let's imagine your organization gave you the `app/internal/admin.py` file.

It contains an `APIRouter` with some admin *path operations* that your organization shares between several projects.

For this example it will be super simple. But let's say that because it is shared with other projects in the organization, we cannot modify it and add a `prefix`, `dependencies`, `tags`, etc. directly to the `APIRouter`:

```
{!../../../docs_src/bigger_applications/app/internal/admin.py!}
```

But we still want to set a custom `prefix` when including the `APIRouter` so that all its *path operations* start with `/admin`, we want to secure it with the `dependencies` we already have for this project, and we want to include `tags` and `responses`.

We can declare all that without having to modify the original `APIRouter` by passing those parameters to `app.include_router()`:

```
{!../../../docs_src/bigger_applications/app/main.py!}
```

That way, the original `APIRouter` will keep unmodified, so we can still share that same `app/internal/admin.py` file with other projects in the organization.

The result is that in our app, each of the *path operations* from the `admin` module will have:

- The prefix `/admin`.
- The tag `admin`.
- The dependency `get_token_header`.
- The response `418`. 🍵

But that will only affect that `APIRouter` in our app, not in any other code that uses it.

So, for example, other projects could use the same `APIRouter` with a different authentication method.

Include a *path operation*

We can also add *path operations* directly to the `FastAPI` app.

Here we do it... just to show that we can 🤷:

```
{!../../../docs_src/bigger_applications/app/main.py!}
```

and it will work correctly, together with all the other *path operations* added with `app.include_router()`.

!!! info "Very Technical Details" **Note**: this is a very technical detail that you probably can **just skip**.

```
---

The `APIRouter`s are not "mounted", they are not isolated from the rest of the
application.

This is because we want to include their *path operations* in the OpenAPI schema and
the user interfaces.

As we cannot just isolate them and "mount" them independently of the rest, the *path
operations* are "cloned" (re-created), not included directly.
```

Check the automatic API docs

Now, run `uvicorn`, using the module `app.main` and the variable `app`:

```
$ uvicorn app.main:app --reload

<span style="color: green;">INFO</span>:     Uvicorn running on
http://127.0.0.1:8000 (Press CTRL+C to quit)
```

And open the docs at http://127.0.0.1:8000/docs.

You will see the automatic API docs, including the paths from all the submodules, using the correct paths (and prefixes) and the correct tags:

Include the same router multiple times with different `prefix`

You can also use `.include_router()` multiple times with the *same* router using different prefixes.

This could be useful, for example, to expose the same API under different prefixes, e.g. `/api/v1` and `/api/latest`.

This is an advanced usage that you might not really need, but it's there in case you do.

Include an `APIRouter` in another

The same way you can include an `APIRouter` in a `FastAPI` application, you can include an `APIRouter` in another `APIRouter` using:

```
router.include_router(other_router)
```

Make sure you do it before including `router` in the `FastAPI` app, so that the *path operations* from `other_router` are also included.