Path Operation Advanced Configuration

OpenAPI operationId

!!! warning If you are not an "expert" in OpenAPI, you probably don't need this.

You can set the OpenAPI `operationId` to be used in your *path operation* with the parameter `operation_id`.

You would have to make sure that it is unique for each operation.

```
Python hl_lines="6" {!../../../docs_src/path_operation_advanced_configuration/tutorial001.py
```

Using the *path operation function* name as the operationId

If you want to use your APIs' function names as `operationId`s, you can iterate over all of them and override each *path operation's* `operation_id` using their `APIRoute.name`.

You should do it after adding all your *path operations*.

```
Python hl_lines="2  12-21  24" {!../../../docs_src/path_operation_advanced_configuration/tut
```

!!! tip If you manually call `app.openapi()`, you should update the `operationId`s before that.

!!! warning If you do this, you have to make sure each one of your *path operation functions* has a unique name.

```
Even if they are in different modules (Python files).
```

Exclude from OpenAPI

To exclude a *path operation* from the generated OpenAPI schema (and thus, from the automatic documentation systems), use the parameter `include_in_schema` and set it to `False`:

```
Python hl_lines="6" {!../../../docs_src/path_operation_advanced_configuration/tutorial003.py
```

Advanced description from docstring

You can limit the lines used from the docstring of a *path operation function* for OpenAPI.

Adding an `\f` (an escaped "form feed" character) causes **FastAPI** to truncate the output used for OpenAPI at this point.

It won't show up in the documentation, but other tools (such as Sphinx) will be able to use the rest.

```
Python hl_lines="19-29" {!../../../docs_src/path_operation_advanced_configuration/tutorial00
```

Additional Responses

You probably have seen how to declare the `response_model` and `status_code` for a *path operation*.

That defines the metadata about the main response of a *path operation*.

You can also declare additional responses with their models, status codes, etc.

There's a whole chapter here in the documentation about it, you can read it at Additional Responses in OpenAPI.

OpenAPI Extra

When you declare a *path operation* in your application, **FastAPI** automatically generates the relevant metadata about that *path operation* to be included in the OpenAPI schema.

!!! note "Technical details" In the OpenAPI specification it is called the Operation Object.

It has all the information about the *path operation* and is used to generate the automatic documentation.

It includes the `tags`, `parameters`, `requestBody`, `responses`, etc.

This *path operation*-specific OpenAPI schema is normally generated automatically by **FastAPI**, but you can also extend it.

!!! tip This is a low level extension point.

`If you only need to declare additional responses, a more convenient way to do it is with [Ad`

You can extend the OpenAPI schema for a *path operation* using the parameter `openapi_extra`.

OpenAPI Extensions

This `openapi_extra` can be helpful, for example, to declare OpenAPI Extensions:

`Python hl_lines="6" {!../../../docs_src/path_operation_advanced_configuration/tutorial005.py`

If you open the automatic API docs, your extension will show up at the bottom of the specific *path operation*.

And if you see the resulting OpenAPI (at `/openapi.json` in your API), you will see your extension as part of the specific *path operation* too:

```
JSON hl_lines="22" {     "openapi": "3.0.2",     "info": {         "title":
"FastAPI",         "version": "0.1.0"     },     "paths": {
"/items/": {             "get": {                 "summary":
"Read Items",                 "operationId": "read_items_items__get",
"responses": {                     "200": {                         "description":
```

```
"Successful Response",                          "content": {
"application/json": {                                  "schema":
{}                              }                         }                       }
},                  "x-aperture-labs-portal": "blue"             }
}     } }
```

Custom OpenAPI *path operation* schema

The dictionary in `openapi_extra` will be deeply merged with the automatically generated OpenAPI schema for the *path operation*.

So, you could add additional data to the automatically generated schema.

For example, you could decide to read and validate the request with your own code, without using the automatic features of FastAPI with Pydantic, but you could still want to define the request in the OpenAPI schema.

You could do that with `openapi_extra`:

```
Python hl_lines="20-37  39-40" {!../../../docs_src/path_operation_advanced_configuration/tut
```

In this example, we didn't declare any Pydantic model. In fact, the request body is not even parsed as JSON, it is read directly as `bytes`, and the function `magic_data_reader()` would be in charge of parsing it in some way.

Nevertheless, we can declare the expected schema for the request body.

Custom OpenAPI content type

Using this same trick, you could use a Pydantic model to define the JSON Schema that is then included in the custom OpenAPI schema section for the *path operation*.

And you could do this even if the data type in the request is not JSON.

For example, in this application we don't use FastAPI's integrated functionality to extract the JSON Schema from Pydantic models nor the automatic validation for JSON. In fact, we are declaring the request content type as YAML, not JSON:

```
Python hl_lines="17-22  24" {!../../../docs_src/path_operation_advanced_configuration/tutori
```

Nevertheless, although we are not using the default integrated functionality, we are still using a Pydantic model to manually generate the JSON Schema for the data that we want to receive in YAML.

Then we use the request directly, and extract the body as `bytes`. This means that FastAPI won't even try to parse the request payload as JSON.

And then in our code, we parse that YAML content directly, and then we are again using the same Pydantic model to validate the YAML content:

```
Python hl_lines="26-33" {!../../../docs_src/path_operation_advanced_configuration/tutorial00
```

!!! tip Here we re-use the same Pydantic model.

`But the same way, we could have validated it in some other way.`