Generate Clients

As **FastAPI** is based on the OpenAPI specification, you get automatic compatibility with many tools, including the automatic API docs (provided by Swagger UI).

One particular advantage that is not necessarily obvious is that you can **generate clients** (sometimes called **SDKs**) for your API, for many different **programming languages**.

OpenAPI Client Generators

There are many tools to generate clients from **OpenAPI**.

A common tool is OpenAPI Generator.

If you are building a **frontend**, a very interesting alternative is openapi-typescript-codegen.

Generate a TypeScript Frontend Client

Let's start with a simple FastAPI application:

=== "Python 3.6 and above"

```
```Python hl_lines="9-11  14-15  18  19  23"
{!> ../../../docs_src/generate_clients/tutorial001.py!}
```
```

=== "Python 3.9 and above"

```
```Python hl_lines="7-9  12-13  16-17  21"
{!> ../../../docs_src/generate_clients/tutorial001_py39.py!}
```
```

Notice that the *path operations* define the models they use for request payload and response payload, using the models `Item` and `ResponseMessage`.

API Docs

If you go to the API docs, you will see that it has the **schemas** for the data to be sent in requests and received in responses:

You can see those schemas because they were declared with the models in the app.

That information is available in the app's **OpenAPI schema**, and then shown in the API docs (by Swagger UI).

And that same information from the models that is included in OpenAPI is what can be used to **generate the client code**.

Generate a TypeScript Client

Now that we have the app with the models, we can generate the client code for the frontend.

Install `openapi-typescript-codegen`

You can install `openapi-typescript-codegen` in your frontend code with:

```
$ npm install openapi-typescript-codegen --save-dev

---> 100%
```

Generate Client Code

To generate the client code you can use the command line application `openapi` that would now be installed.

Because it is installed in the local project, you probably wouldn't be able to call that command directly, but you would put it on your `package.json` file.

It could look like this:

```
{
  "name": "frontend-app",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "generate-client": "openapi --input http://localhost:8000/openapi.json --output
./src/client --client axios"
  },
  "author": "",
  "license": "",
  "devDependencies": {
    "openapi-typescript-codegen": "^0.20.1",
    "typescript": "^4.6.2"
  }
}
```

After having that NPM `generate-client` script there, you can run it with:

```
$ npm run generate-client

frontend-app@1.0.0 generate-client /home/user/code/frontend-app
> openapi --input http://localhost:8000/openapi.json --output ./src/client --client
axios
```

That command will generate code in `./src/client` and will use `axios` (the frontend HTTP library) internally.

Try Out the Client Code

Now you can import and use the client code, it could look like this, notice that you get autocompletion for the methods:

You will also get autocompletion for the payload to send:

!!! tip Notice the autocompletion for `name` and `price`, that was defined in the FastAPI application, in the `Item` model.

You will have inline errors for the data that you send:

The response object will also have autocompletion:

FastAPI App with Tags

In many cases your FastAPI app will be bigger, and you will probably use tags to separate different groups of *path operations*.

For example, you could have a section for **items** and another section for **users**, and they could be separated by tags:

=== "Python 3.6 and above"

```
```Python hl_lines="23  28  36"
{!> ../../../docs_src/generate_clients/tutorial002.py!}
```
```

=== "Python 3.9 and above"

```
```Python hl_lines="21  26  34"
{!> ../../../docs_src/generate_clients/tutorial002_py39.py!}
```
```

Generate a TypeScript Client with Tags

If you generate a client for a FastAPI app using tags, it will normally also separate the client code based on the tags.

This way you will be able to have things ordered and grouped correctly for the client code:

In this case you have:

- `ItemsService`
- `UsersService`

Client Method Names

Right now the generated method names like `createItemItemsPost` don't look very clean:

```
ItemsService.createItemItemsPost({name: "Plumbus", price: 5})
```

...that's because the client generator uses the OpenAPI internal **operation ID** for each *path operation*.

OpenAPI requires that each operation ID is unique across all the *path operations*, so FastAPI uses the **function name**, the **path**, and the **HTTP method/operation** to generate that operation ID, because that way it can make sure that the operation IDs are unique.

But I'll show you how to improve that next. 🤓

Custom Operation IDs and Better Method Names

You can **modify** the way these operation IDs are **generated** to make them simpler and have **simpler method names** in the clients.

In this case you will have to ensure that each operation ID is **unique** in some other way.

For example, you could make sure that each *path operation* has a tag, and then generate the operation ID based on the **tag** and the *path operation* **name** (the function name).

Custom Generate Unique ID Function

FastAPI uses a **unique ID** for each *path operation*, it is used for the **operation ID** and also for the names of any needed custom models, for requests or responses.

You can customize that function. It takes an `APIRoute` and outputs a string.

For example, here it is using the first tag (you will probably have only one tag) and the *path operation* name (the function name).

You can then pass that custom function to **FastAPI** as the `generate_unique_id_function` parameter:

=== "Python 3.6 and above"

```
```Python hl_lines="8-9  12"
{!> ../../../docs_src/generate_clients/tutorial003.py!}
```
```

=== "Python 3.9 and above"

```
```Python hl_lines="6-7  10"
{!> ../../../docs_src/generate_clients/tutorial003_py39.py!}
```
```

Generate a TypeScript Client with Custom Operation IDs

Now if you generate the client again, you will see that it has the improved method names:

As you see, the method names now have the tag and then the function name, now they don't include information from the URL path and the HTTP operation.

Preprocess the OpenAPI Specification for the Client Generator

The generated code still has some **duplicated information**.

We already know that this method is related to the **items** because that word is in the `ItemsService` (taken from the tag), but we still have the tag name prefixed in the method name too. 😕

We will probably still want to keep it for OpenAPI in general, as that will ensure that the operation IDs are **unique**.

But for the generated client we could **modify** the OpenAPI operation IDs right before generating the clients, just to make those method names nicer and **cleaner**.

We could download the OpenAPI JSON to a file `openapi.json` and then we could **remove that prefixed tag** with a script like this:

```
{!../../../docs_src/generate_clients/tutorial004.py!}
```

With that, the operation IDs would be renamed from things like `items-get_items` to just `get_items`, that way the client generator can generate simpler method names.

Generate a TypeScript Client with the Preprocessed OpenAPI

Now as the end result is in a file `openapi.json`, you would modify the `package.json` to use that local file, for example:

```
{
  "name": "frontend-app",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "generate-client": "openapi --input ./openapi.json --output ./src/client --
client axios"
  },
  "author": "",
  "license": "",
  "devDependencies": {
    "openapi-typescript-codegen": "^0.20.1",
    "typescript": "^4.6.2"
  }
}
```

After generating the new client, you would now have **clean method names**, with all the **autocompletion**, **inline errors**, etc:

Benefits

When using the automatically generated clients you would **autocompletion** for:

- Methods.
- Request payloads in the body, query parameters, etc.
- Response payloads.

You would also have **inline errors** for everything.

And whenever you update the backend code, and **regenerate** the frontend, it would have any new *path operations* available as methods, the old ones removed, and any other change would be reflected on the generated code. 🤓

This also means that if something changed it will be **reflected** on the client code automatically. And if you **build** the client it will error out if you have any **mismatch** in the data used.

So, you would **detect many errors** very early in the development cycle instead of having to wait for the errors to show up to your final users in production and then trying to debug where the problem is. ✨