最初のステップ

最もシンプルなFastAPIファイルは以下のようになります:

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

これを `main.py` にコピーします。

ライブサーバーを実行します:

```
$ uvicorn main:app --reload

<span style="color: green;">INFO</span>:     Uvicorn running on
http://127.0.0.1:8000 (Press CTRL+C to quit)
<span style="color: green;">INFO</span>:     Started reloader process [28720]
<span style="color: green;">INFO</span>:     Started server process [28722]
<span style="color: green;">INFO</span>:     Waiting for application startup.
<span style="color: green;">INFO</span>:     Application startup complete.
```

!!! note "備考" `uvicorn main:app` は以下を示します:

```
* `main`: `main.py`ファイル (Python "module")。
* `app`:  `main.py`内部で作られるobject (`app = FastAPI()`のように記述される)。
* `--reload`: コードの変更時にサーバーを再起動させる。開発用。
```

出力には次のような行があります:

```
INFO:     Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
```

この行はローカルマシンでアプリが提供されているURLを示しています。

チェック

ブラウザでhttp://127.0.0.1:8000を開きます。

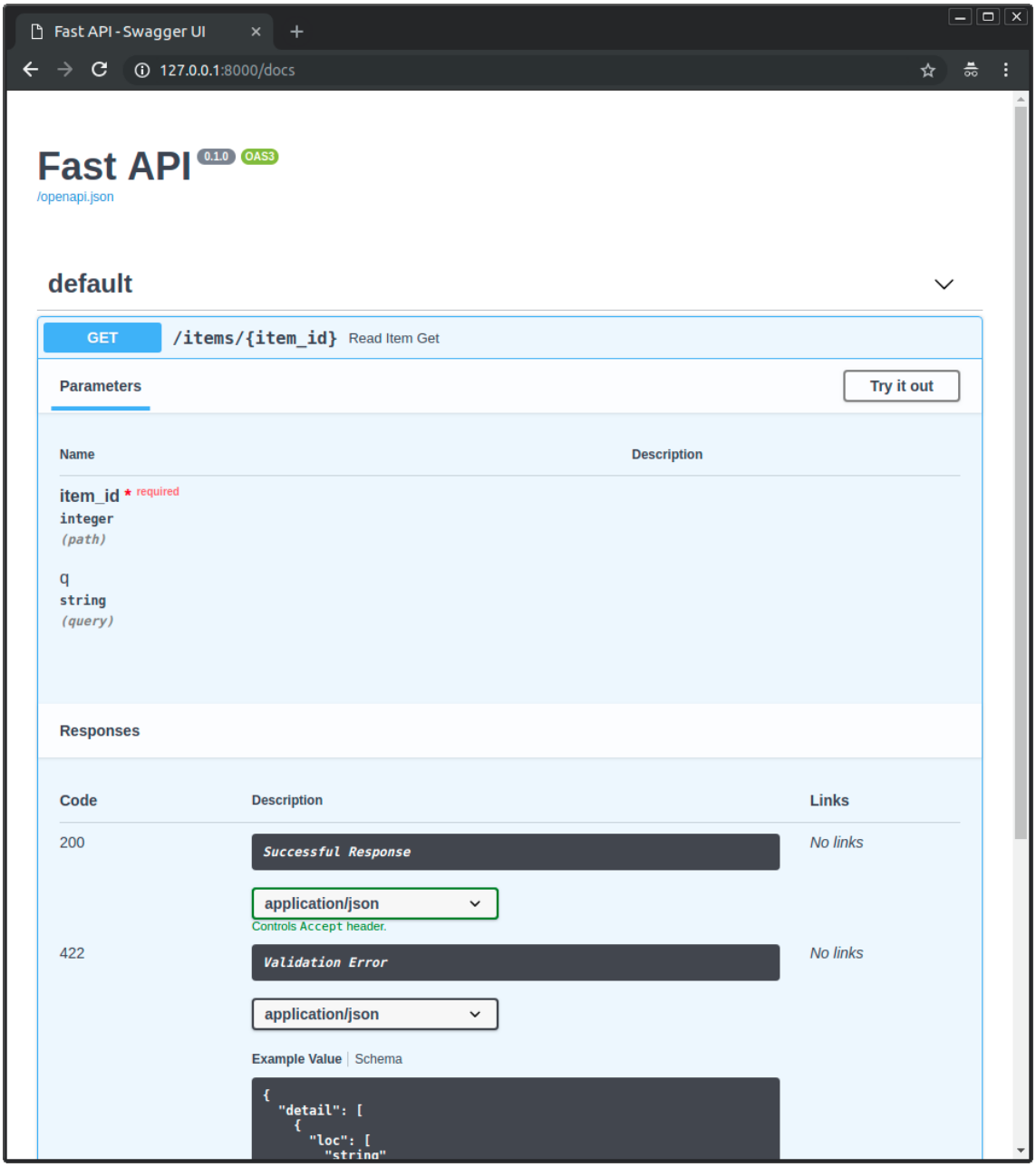
次のようなJSONレスポンスが表示されます:

```
{"message": "Hello World"}
```

対話的APIドキュメント

次に、http://127.0.0.1:8000/docsにアクセスします。

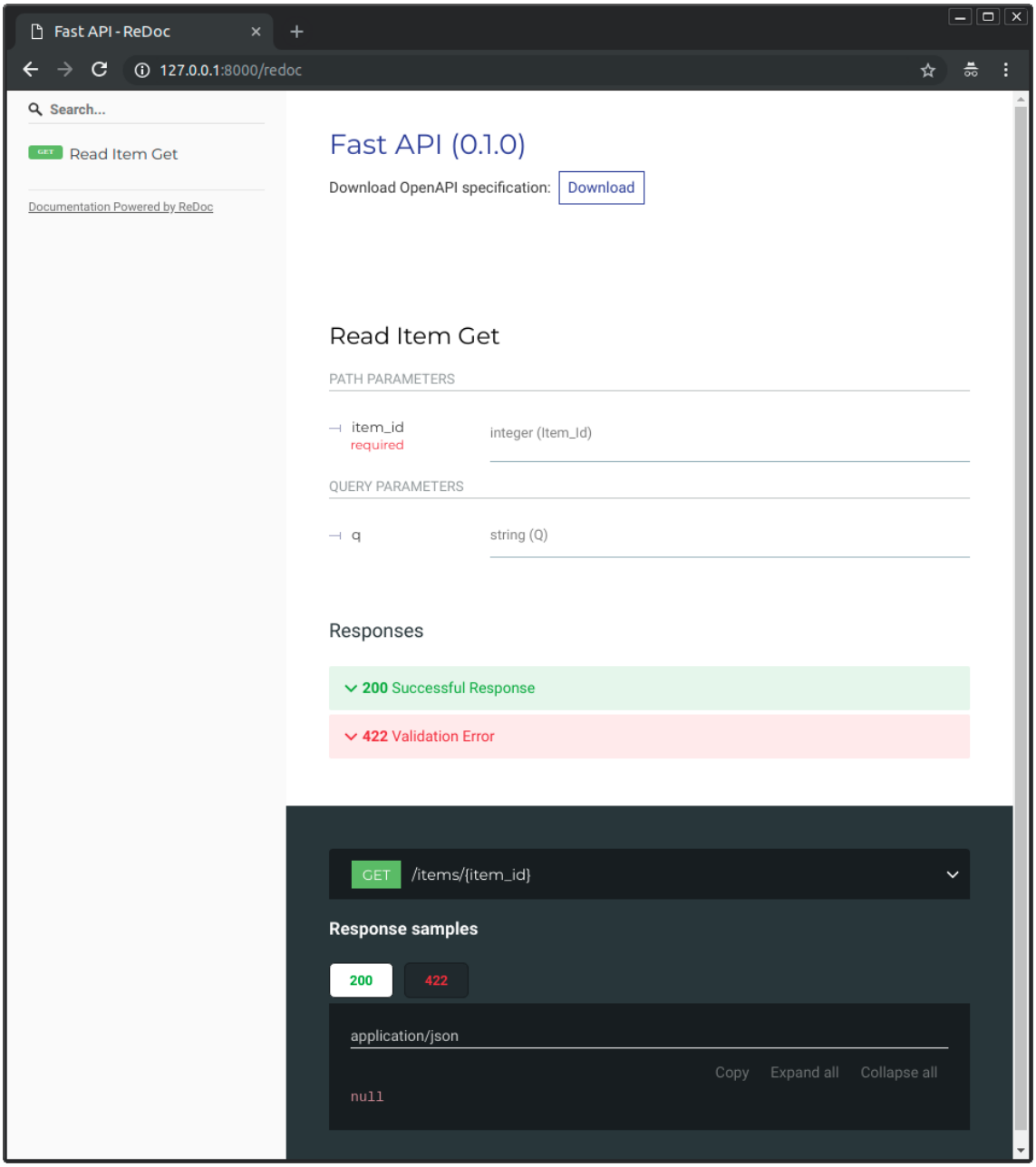
自動生成された対話的APIドキュメントが表示されます (Swagger UIで提供):



他のAPIドキュメント

次に、http://127.0.0.1:8000/redocにアクセスします。

先ほどとは異なる、自動生成された対話的APIドキュメントが表示されます (ReDocによって提供):



OpenAPI

FastAPIは、APIを定義するための**OpenAPI**標準規格を使用して、すべてのAPIの「スキーマ」を生成します。

「スキーマ」

「スキーマ」は定義または説明です。実装コードではなく、単なる抽象的な説明です。

API「スキーマ」

ここでは、OpenAPIはAPIのスキーマ定義の方法を規定する仕様です。

このスキーマ定義はAPIパス、受け取り可能なパラメータなどが含まれます。

データ「スキーマ」

「スキーマ」という用語は、JSONコンテンツなどの一部のデータの形状を指す場合もあります。

そのような場合、スキーマはJSON属性とそれらが持つデータ型などを意味します。

OpenAPIおよびJSONスキーマ

OpenAPIはAPIのためのAPIスキーマを定義します。そして、そのスキーマは**JSONデータスキーマ**の標準規格に準拠したJSONスキーマを利用するAPIによって送受されるデータの定義（または「スキーマ」）を含んでいます。

`openapi.json` を確認

素のOpenAPIスキーマがどのようなものか興味がある場合、FastAPIはすべてのAPIの説明を含むJSON（スキーマ）を自動的に生成します。

次の場所で直接確認できます: http://127.0.0.1:8000/openapi.json.

次のようなJSONが表示されます。

```
{
    "openapi": "3.0.2",
    "info": {
        "title": "FastAPI",
        "version": "0.1.0"
    },
    "paths": {
        "/items/": {
            "get": {
                "responses": {
                    "200": {
                        "description": "Successful Response",
                        "content": {
                            "application/json": {


...
```

OpenAPIの目的

OpenAPIスキーマは、FastAPIに含まれている2つのインタラクティブなドキュメントシステムの動力源です。

そして、OpenAPIに基づいた代替案が数十通りあります。 **FastAPI**で構築されたアプリケーションに、これらの選択肢を簡単に追加できます。

また、APIと通信するクライアント用のコードを自動的に生成するために使用することもできます。たとえば、フロントエンド、モバイル、またはIoTアプリケーションです。

ステップ毎の要約

Step 1: `FastAPI` をインポート

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

`FastAPI` は、APIのすべての機能を提供するPythonクラスです。

!!! note "技術詳細" `FastAPI` は `Starlette` を直接継承するクラスです。

```
`FastAPI`でも<a href="https://www.starlette.io/" class="external-link"
target="_blank">Starlette</a>のすべての機能を利用可能です。
```

Step 2: `FastAPI` の「インスタンス」を生成

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

ここで、 `app` 変数が `FastAPI` クラスの「インスタンス」になります。

これが、すべてのAPIを作成するための主要なポイントになります。

この `app` はコマンドで `uvicorn` が参照するものと同じです:

```
$ uvicorn main:app --reload

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

以下のようなアプリを作成したとき:

```
{!../../../docs_src/first_steps/tutorial002.py!}
```

そして、それを `main.py` ファイルに置き、次のように `uvicorn` を呼び出します:

```
$ uvicorn main:my_awesome_api --reload

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

Step 3: *path operation*を作成

パス

ここでの「パス」とは、最初の `/` から始まるURLの最後の部分を指します。

したがって、次のようなURLでは:

```
https://example.com/items/foo
```

...パスは次のようになります:

```
/items/foo
```

!!! info "情報" 「パス」は一般に「エンドポイント」または「ルート」とも呼ばれます。

APIを構築する際、「パス」は「関心事」と「リソース」を分離するための主要な方法です。

Operation

ここでの「オペレーション」とは、HTTPの「メソッド」の1つを指します。

以下のようなものの1つ:

- `POST`
- `GET`
- `PUT`
- `DELETE`

...さらによりエキゾチックなもの:

- `OPTIONS`
- `HEAD`
- `PATCH`
- `TRACE`

HTTPプロトコルでは、これらの「メソッド」の1つ（または複数）を使用して各パスにアクセスできます。

APIを構築するときは、通常、これらの特定のHTTPメソッドを使用して特定のアクションを実行します。

通常は次を使用します:

- `POST`：データの作成
- `GET`：データの読み取り
- `PUT`：データの更新
- `DELETE`：データの削除

したがって、OpenAPIでは、各HTTPメソッドは「オペレーション」と呼ばれます。

「**オペレーションズ**」とも呼ぶことにします。

パスオペレーションデコレータを定義

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

`@app.get("/")` は直下の関数が下記のリクエストの処理を担当することを**FastAPI**に伝えます:

- パス `/`
- `get` オペレーション

!!! info " `@decorator` について" Pythonにおける `@something` シンタックスはデコレータと呼ばれます。

「デコレータ」は関数の上に置きます。かわいらしい装飾的な帽子のようです（この用語の由来はそこにあると思います）。

「デコレータ」は直下の関数を受け取り、それを使って何かを行います。

私たちの場合、このデコレーターは直下の関数が**オペレーション** `get`を使用した**パス**` / `に対応することを**FastAPI** に通知します。

これが「*パスオペレーションデコレータ*」です。

他のオペレーションも使用できます:

- `@app.post()`
- `@app.put()`
- `@app.delete()`

また、よりエキゾチックなものも使用できます:

- `@app.options()`
- `@app.head()`
- `@app.patch()`
- `@app.trace()`

!!! tip "豆知識" 各オペレーション (HTTPメソッド)は自由に使用できます。

```
**FastAPI**は特定の意味づけを強制しません。

ここでの情報は、要件ではなくガイドラインとして提示されます。

例えば、GraphQLを使用する場合、通常は`POST`オペレーションのみを使用してすべてのアクションを実行します。
```

Step 4: パスオペレーションを定義

以下は「**パスオペレーション関数**」です:

- **パス**: は `/` です。
- **オペレーション**: は `get` です。
- **関数**: 「デコレータ」の直下にある関数 (`@app.get("/")` の直下) です。

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

これは、Pythonの関数です。

この関数は、 `GET` オペレーションを使ったURL「 `/` 」へのリクエストを受け取るたびに**FastAPI**によって呼び出されます。

この場合、この関数は `async` 関数です。

`async def` の代わりに通常の関数として定義することもできます:

```
{!../../../docs_src/first_steps/tutorial003.py!}
```

!!! note "備考" 違いが分からない場合は、Async: *"In a hurry?"*{.internal-link target=_blank}を確認してください。

Step 5: コンテンツの返信

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

`dict` 、 `list` 、 `str` 、 `int` などを返すことができます。

Pydanticモデルを返すこともできます（後で詳しく説明します）。

JSONに自動的に変換されるオブジェクトやモデルは他にもたくさんあります（ORMなど）。 お気に入りのものを使ってみてください。すでにサポートされている可能性が高いです。

まとめ

- `FastAPI` をインポート
- `app` インスタンスを生成
- **パスオペレーションデコレータ**を記述 (`@app.get("/")`)
- **パスオペレーション関数**を定義 (上記の `def root(): ...` のように)
- 開発サーバーを起動 (`uvicorn main:app --reload`)