


The background is a vibrant, abstract composition of light trails in shades of blue, teal, and yellow, creating a sense of motion and depth. A large, white, rounded rectangular box is centered in the image, serving as a backdrop for the text.

# FastAPI その1



# この資料について



オンライン学習プラットフォーム『Udemy』  
で公開している、  
『Python x FastAPI初心者向け講座』の  
説明用資料です

[https://www.udemy.com/course/  
python\\_fastapi](https://www.udemy.com/course/python_fastapi)





# FastAPIの概要



# フレームワーク (枠組み)



Webアプリ、機械学習・ディープラーニングのサービスを開発する際の土台  
必要な機能、よく使われる機能をまとめて  
使いやすくしたもの

# Python フレームワーク

フレームワーク	Flask	Django	FastAPI
特徴	軽量	フルスタックで高機能	高速で API開発に特化
メリット	シンプルなアプリから はじめやすい 拡張しやすい	豊富な機能が元々備わっている セキュリティ高い 大規模開発向け	高速なパフォーマンス 非同期処理が簡単にできる 自動的なAPIドキュメント生成
デメリット	大規模アプリになると 手動で多くの設定が必要	軽量なアプリでは オーバースペックになる場合がある	非同期処理の学習が必要 新しいフレームワークなので 情報が少ない

# FastAPI登場の背景



2018年 リリース オープンソース Python 3.8+

## 1. モダンWebアプリの需要

バックエンドのAPIが高速で、効率的で、拡張性が必要

## 2. 非同期処理

Python3.5以降 asyncioライブラリが標準化

## 3. API開発の標準化・自動化

Restful APIやGraphQLなどのAPI設計が一般化、ドキュメント自動生成の重要性アップ

## 4. 型安全性と開発速度の向上

Python3.6以降 型ヒント コードの可読性・保守性・生産性アップ

## 5. マイクロサービスアーキテクチャ

小さく独立したサービスを開発し、それらを組み合わせて大規模アプリを構築するアプローチが増えた。(Go言語など)

# 環境構築 1



任意のフォルダで仮想環境作成

mac

/Users/{ユーザー名}/python/fastapi-test

win

C:¥python¥fastapi-test

# 環境構築 2



仮想環境の作成 (-mはモジュール名指定)

```
$ python -m venv .venv
```

有効化

```
$ . .venv/bin/activate
```

```
(.venv) $
```

終了

```
(.venv) $ deactivate
```

```
$
```






# Gitリポジトリ



# Gitリポジトリ



この講座は初心者向けという事で  
gitは使わなくてもいいように進めます。

講座受講中にトラブルなどあり、  
コード確認を依頼される場合があれば  
githubなどで共有いただけると楽ではあります。  
(github使わない場合はコードをzip圧縮してメール送信)



# GitHub



リポジトリURL

[https://github.com/aokitashipro/  
udemy\\_fastapi\\_basic](https://github.com/aokitashipro/udemy_fastapi_basic)





# FastAPI インストール



# インストール

公式ホームページ

<https://fastapi.tiangolo.com/ja/>

インストール

```
$ pip install fastapi==0.110.0
```

```
$ pip install "uvicorn[standard]"==0.27.1
```

一覧表示

```
$ pip list
```

# インストールされたパッケージ (抜粋)

annotated-types # 型ヒントの拡張

anyio # 非同期を簡単にする

click # コマンドラインアプリ用

fastapi # フレームワーク

h11 # HTTPを使うライブラリ

idna # インターネットドメインを処理

pip # パッケージ管理

pydantic # データバリデーション、設定管理

pydantic\_core # pydanticのコア

sniffio # 現在実行中の非同期ライブラリを識別

starlette # FastAPIの基盤フレームワーク

typing\_extensions # typingモジュールの拡張

uvicorn # 高速な非同期特化のサーバ



# 最初の一歩



**main.py**

```
from fastapi import FastAPI
```


```
app = FastAPI()
```

```
@app.get("/")
```

```
async def root():
```

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

# サーバー起動



非同期用の簡易サーバー

uvicorn (ユビコーン ASGI Webサーバ)

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

ブラウザで

<http://127.0.0.1:8000> レスpons表示

<http://127.0.0.1:8000/docs> 自動生成 対話的APIドキュメント表示(Swagger UI)

<http://127.0.0.1:8000/redoc> 自動生成 対話的APIドキュメント表示(ReDoc)

サーバーを止めるには Ctrl + C



# コードの解説



**main.py**

```
from fastapi import FastAPI #パッケージのインポート
```

```
app = FastAPI() # クラスをインスタンス化
```

```
@app.get("/") # インスタンス内のメソッドをデコレータで指定
```

```
async def root(): # パスオペレーション関数 asyncは非同期
```

```
    return {"message": "Hello World"} # 辞書型で返信
```





# ルーティング



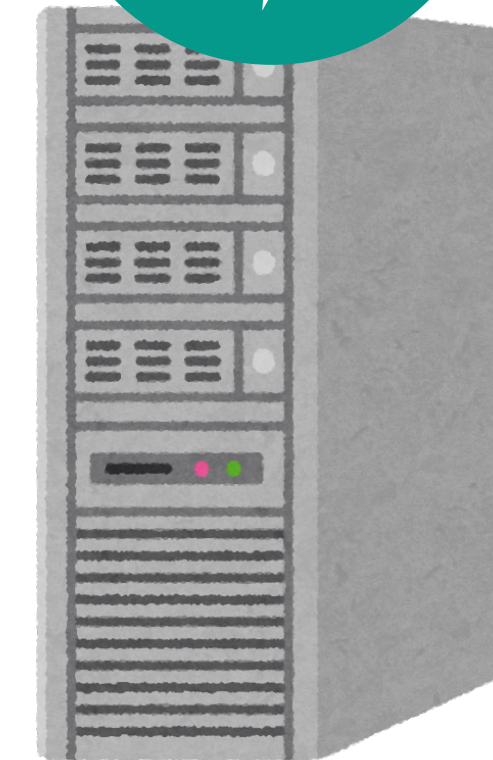
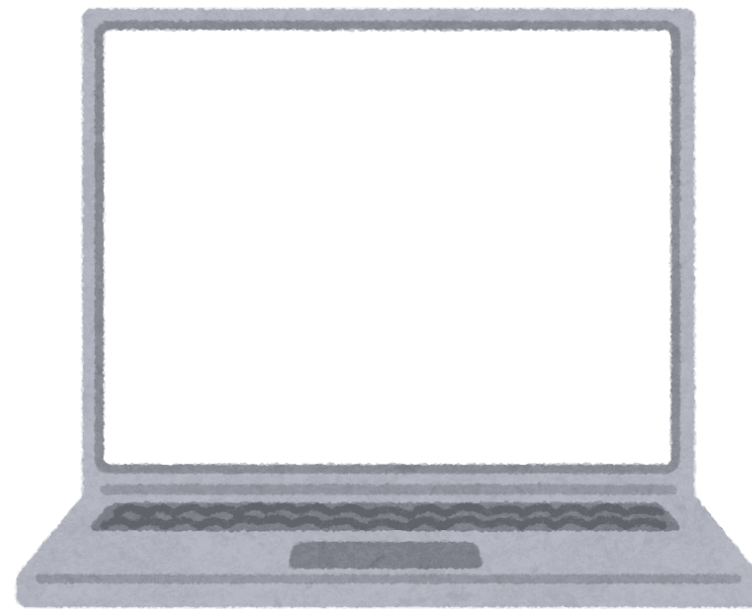
# ルーティング

URLに応じて返答を返す

パス、エンドポイント、ルートなどいろいろな呼び方がある

ドメイン/パス(エンドポイント、ルート)

`https://yahoo.co.jp/xxx`



JSON



# CRUD



Create ・ ・ 新規作成

Read ・ ・ 読み込み(表示)

Update ・ ・ 更新

Delete ・ ・ 削除



# RESTful API (設計原則・考え方)

HTTPメソッド	URI	CRUD	用途
get	/items	Read	一覧表示
get	/items/create	Create	新規作成 (UI向けAPIでは不要)
post	/items	Create	保存
get	/items/{id}	Read	詳細表示
get	/items/{id}/edit	Read	詳細編集 (UI向けAPIでは不要)
put/patch	/items/{id}	Update	更新
delete	/items/{id}	Delete	削除

# パスオペレーション関数



メソッドとパスを指定して処理できる仕組み

`@app.get('/')`

`@app.get('/items')`

`@app.post('/')`

他に `@app.put()`, `@app.delete()` などもある



# APIRouter



FastAPIクラスでルーティング

アプリ全体のルーティングを一箇所で管理

小規模アプリ、シンプルなAPI向き

APIRouterクラスでルーティング

複数のモジュールに分割して管理

大規模アプリや整理された構造のAPI開発向き

モジュールを `include_router()` で読み込むことができる

(マニュアル [Reference/APIRouter class](#))

# フォルダ構成



FastAPIは自動でドキュメント生成

先にルーティングの雛形を作っておくと便利

フォルダ構成は自由

main.py

routers/

\_\_init\_\_.py # 空のファイル パッケージ化対応

contact.py



## ルーティング routers/contact.py

```
from fastapi import APIRouter # ルーティング設定用のクラス  
router = APIRouter() # インスタンス化
```

```
@router.get("/contacts") # 一覧表示  
async def get_contact_all():  
    pass
```

```
@router.post("/contacts") # 保存  
async def create_contact():  
    pass
```

# ルーティング routers/contact.py

```
@router.get("/contacts/{id}") # 詳細表示
```

```
async def get_contact():
```

```
    pass
```

```
@router.put("/contacts/{id}") # 更新
```

```
async def update_contact():
```

```
    pass
```

```
@router.delete("/contacts/{id}") # 削除
```

```
async def delete_contact():
```

```
    pass
```



# main.py 書き換え



```
from fastapi import FastAPI
from routers import contact # パッケージ読み込

app = FastAPI()

# パッケージ内のルーター(インスタンス)を読み込み
app.include_router(contact.router)
```

# ブラウザで確認

<http://127.0.0.1:8000>

**FastAPI** 0.1.0 OAS 3.1

</openapi.json>

default

^

GET

/contacts Index

▼

POST

/contacts Store

▼

GET

/contacts/{id} Show

▼

PUT

/contacts/{id} Update

▼

DELETE

/contacts/{id} Delete

▼





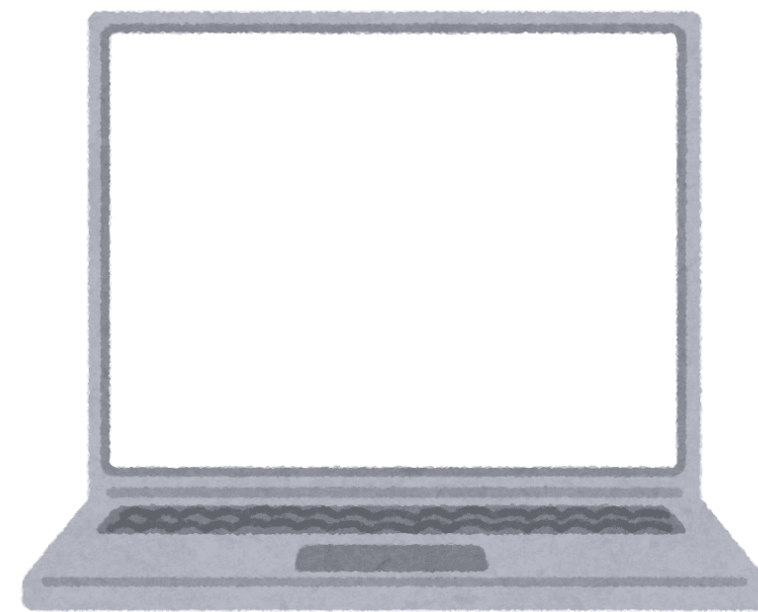
# レスポンス (スキーマ)



# リクエストとレスポンス

実運用では

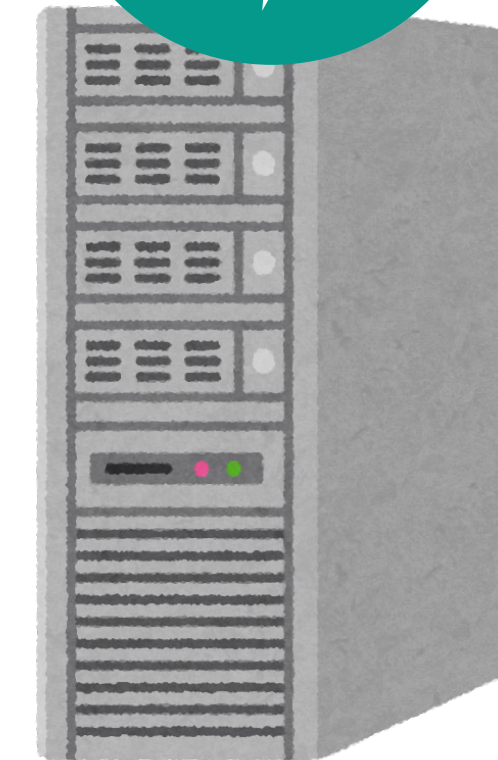
データベースにデータ保存される



リクエスト

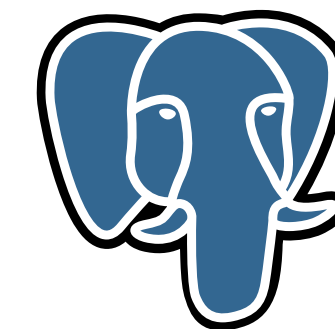


レスポンス



MySQL™

SQLite





# 最終的なテーブル構成

お問い合わせフォームを想定

論理	物理 (列名)	データ型	キーなど
id	id	int	PK
氏名	name	string	
メール	email	string	
Url	url	string	nullable
性別	gender	int	
問い合わせ内容	message	string	
同意チェック	is_enabled	boolean	
登録日時	created_at	datetime	

# スキーマとモデル



スキーマ(定義・説明用)

API通信用のデータモデル (バリデーションなど)

Pydanticで作成

データベース接続(ORM)のモデル (テーブル構成)

SQLAlchemyで作成



# スキーマ(定義・説明用)



Pydantic ・ ・ データの検証(チェック)と設定管理

BaseModelを継承して使う

Field ・ ・ バリデーション、デフォルト値など設定ができる

<https://docs.pydantic.dev/2.6/concepts/fields/>

# スキーマ(簡易バリデーション)

`schemas/contact.py`

```
from pydantic import BaseModel, Field #インポート
```

```
from datetime import datetime
```

```
class Contact(BaseModel): # 継承
```

```
    id: int
```

```
    name: str
```

```
    email: str
```

```
    url: str
```

```
    gender: int
```

```
    message: str
```

```
    is_enabled: bool
```

```
    created_at: datetime
```



# ルート情報の変更

`routers/contact.py`

```
from fastapi import APIRouter
```

```
import schemas.contact as contact_schema # 追加 (後ほどデータモデルも扱うのでschemaと記載)
```

```
from datetime import datetime
```

```
router = APIRouter()
```

```
# 一覧表示
```

```
# 第二引数にレスポンスのモデルを指定
```

```
@router.get("/contacts", response_model=list[contact_schema.Contact])
```

```
async def get_contact_all():
```

```
    # 試しにデータを登録
```

```
    dummy_date = datetime.now()
```

```
    return [contact_schema.Contact(id=1, name="山田", email="test@test.com",  
url="http://test.com", gender=1, message="テスト", is_enabled=False,  
created_at=dummy_date )]
```

# SwaggerUI

GET

/contacts Index

Try it out

Parameters

No parameters

Responses

Code	Description	Links
200	<div>Successful Response</div> <div>Media type</div> <div>application/json</div> <div>Controls Accept header.</div> <div>Example Value   Schema</div> <pre>[   {     "id": 0,     "name": "string",     "email": "string",     "url": "string",     "gender": 0,     "message": "string",     "is_enabled": false   } ]</pre>	No links

Execute  
をクリック

GET

/contacts Index

Cancel

Parameters

No parameters

Execute

Clear

Responses

Curl

```
curl -X 'GET' \
  'http://127.0.0.1:8000/contacts' \
  -H 'accept: application/json'
```

Request URL

http://127.0.0.1:8000/contacts

Server response

Code	Details
200	<div>Response body</div> <pre>[   {     "id": 1,     "name": "山田",     "email": "test@test.com",     "url": null,     "gender": 1,     "message": "テスト",     "is_enabled": false   } ]</pre> <div>Download</div>



# 型が違っているとエラー

試しに schemas/contact.pyの  
emailの型をintに変えてみる

SwaggerUIで実行すると  
サーバーエラーが発生

Responses

Curl

```
curl -X 'GET' \  
  'http://127.0.0.1:8000/contacts' \  
  -H 'accept: application/json'
```

Request URL

```
http://127.0.0.1:8000/contacts
```

Server response

Code	Details
500 <i>Undocumented</i>	Error: Internal Server Error

Response body

```
Internal Server Error
```

Download

# スキーマ(バリデーション)

\$ pip install email\_validator==2.1.1 # メールバリデーション(内部的に必要)

**schemas/contact.py**

```
from datetime import datetime
```

```
from pydantic import BaseModel, Field, EmailStr, HttpUrl # 追加
```

```
class Contact(BaseModel):
```

```
    id: int
```

```
    name: str = Field(..., min_length=2, max_length=50) # 必須, 2文字~50文字
```

```
    email: EmailStr # メール
```

```
    url: HttpUrl | None = Field(default=None) # urlか空
```

```
    gender: int = Field(..., strict=True, ge=0, le=2) # 必須, 0, 1, 2
```

```
    message: str = Field(..., max_length=200) # 必須、最大200文字
```

```
    is_enabled: bool = Field(default=False) # デフォルト値指定
```

```
    created_at: datetime
```





# リクエスト (スキーマ)



# リクエスト (POST通信)

リクエストボディに情報を含め送信

**routers/contact.py**

# 保存

# 第二引数にモデルを設定

@router.post("/contacts",

**response\_model**=contact\_schema.Contact)

# 引数にモデル指定 model\_dump()は辞書生成

async def create\_contact(body: contact\_schema.Contact):

return contact\_schema.Contact(\*\*body.model\_dump())



# SwaggerUI

**POST** /contacts Store ^

Parameters Try it out

No parameters

Request body **required** application/json ▼

Example Value | Schema

```
{
  "id": 0,
  "name": "string",
  "email": "string",
  "url": "string",
  "gender": 0,
  "message": "string",
  "is_enabled": false
}
```

Responses

Code	Description	Links
200	Successful Response	No links

Try it outをクリック後  
Request bodyを編集して  
Execute実行する

Request body **required** application/json ▼

```
{
  "id": 1,
  "name": "田中",
  "email": "test123@test.com",
  "url": "http://test.com",
  "gender": 1,
  "message": "メッセージ",
  "is_enabled": true
}
```

Execute

# リクエスト→レスポンス

リクエストした内容が  
そのまま  
レスポンスとして  
返ってくればOK

## Responses

### Curl

```
curl -X 'POST' \  
  'http://127.0.0.1:8000/contacts' \  
  -H 'accept: application/json' \  
  -H 'Content-Type: application/json' \  
  -d '{  
    "id": 1,  
    "name": "田中",  
    "email": "test123@test.com",  
    "url": "http://test.com",  
    "gender": 1,  
    "message": "メッセージ",  
    "is_enabled": true  
  }'
```

### Request URL

http://127.0.0.1:8000/contacts

### Server response

#### Code

#### Details

200

#### Response body

```
{  
  "id": 1,  
  "name": "田中",  
  "email": "test123@test.com",  
  "url": "http://test.com",  
  "gender": 1,  
  "message": "メッセージ",  
  "is_enabled": true  
}
```



Download





# API通信して 情報取得してみる



# API通信してみる

```
$ pip install requests # インストール
```

```
api_test.py import requests, import json, from datetime import datetime
```

```
def main():
```

```
    url = "http://localhost:8000/contacts"
```

```
    current_datetime = datetime.now().isoformat() # JSON変換できるようにISO形式に変換
```

```
    body = { "id": 1, "name": "string", "email": "user@example.com", "url": "https://example.com/",  
            "gender": 0, "message": "string", "is_enabled": False, "created_at" : current_datetime  
    }
```

```
    # 辞書型->JSONに変換してPOST通信
```

```
    res = requests.post(url, json.dumps(body))
```

```
    print(res.json())
```

```
if __name__ == "__main__": # 直接実行された時だけ動く (__name__ 変数がmainとして生成)
```

```
    main()
```



# POST通信してみる



```
$ python api_test.py
```

```
{'id': 1, 'name': 'string', 'email': 'user@example.com',  
'url': 'https://example.com/', 'gender': 0, 'message':  
'string', 'is_enabled': False,  
'created_at': '2024-01-01...'}
```

リクエストボディで送信した値が  
返って来ればOK





# その他の ルート設定



# 他のルート設定

ルートパラメータにidを持つ。引数にidを渡す

# 詳細表示

```
@router.get("/contacts/{id}", response_model=contact_schema.Contact)
```

```
async def get_contact(id: int):
```

```
    return contact_schema.Contact(id)
```

# 更新

```
@router.put("/contacts/{id}", response_model=contact_schema.Contact)
```

```
async def update_contact(id: int, body: contact_schema.Contact):
```

```
    return contact_schema.Contact(id, **body.model_dump())
```

# 削除

```
@router.delete("/contacts/{id}", response_model=contact_schema.Contact)
```

```
async def delete_contact(id: int):
```

```
    return
```

# スキーマを作るメリット



APIモック(叩き台)としての役割

APIマニュアル生成済み

フロントエンド担当にも簡単に情報共有できる

ルータとスキーマを肉付けしていけば

API機能が実装できる

機能修正時・・・マニュアル自動更新なので安心